Domain 3: Prepare and process data with Azure Databricks

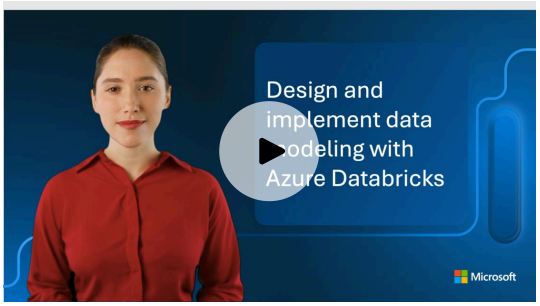
Design and implement data modeling with Azure Databricks

1. Introduction

https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/1-introduction

Introduction

Completed



Your organization processes millions of transactions daily from multiple source systems, and stakeholders need reliable analytics that reflect both current operations and historical trends. The data engineering team faces critical decisions: How should data flow from source systems into the **lakehouse**? Which table formats provide the right balance of performance and compatibility? How do you maintain historical accuracy while keeping queries fast?

Data modeling in Azure Databricks with **Unity Catalog** addresses these challenges through deliberate design choices that affect every downstream consumer. You select **ingestion patterns** that match your latency requirements. You choose table formats like **Delta Lake** or **Apache Iceberg** based on transaction needs and cross-platform compatibility. You design **partitioning schemes** and **clustering strategies** that align with how analysts query the data.

These decisions compound over time. A table partitioned incorrectly today creates performance problems that grow with data volume. A **dimension table** without proper change tracking loses historical context. Thoughtful upfront design creates a foundation that scales efficiently and answers questions you haven't yet anticipated.

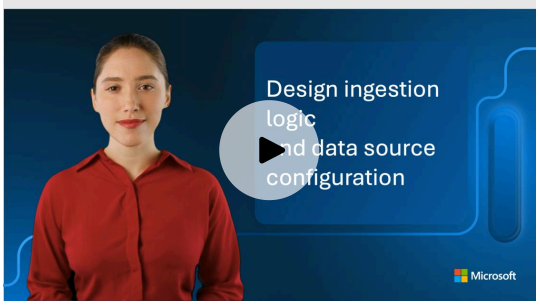
This module guides you through key data modeling decisions for Azure Databricks environments. You'll learn to design ingestion logic, select the right tools for each data source type, and implement storage strategies that optimize query performance. You'll explore **slowly changing dimensions** for historical accuracy and understand when **managed tables** provide advantages over **external tables**.

2. Design ingestion logic and data source configuration

https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/2-design-data-ingestion-logic

Design ingestion logic and data source configuration

Completed



Designing ingestion logic requires you to make critical decisions about how data moves from source systems into Azure Databricks. These decisions affect data freshness, processing efficiency, and system reliability. Before you select tools or write code, you need to understand the characteristics of your data sources and define extraction strategies that align with your business requirements.

In this unit, you explore the key design considerations for data ingestion logic and data source configuration in Azure Databricks. The focus here is on conceptual logic and design patterns—specific tool selection and implementation details are covered in later units.

Understand extraction types

Extraction type determines how you capture data from source systems and impacts both the **timeliness** and **efficiency** of your data pipeline. Azure Databricks supports three primary extraction patterns, each suited to different scenarios.

Full extraction

Full extraction reads the entire dataset from the source system during each ingestion run.

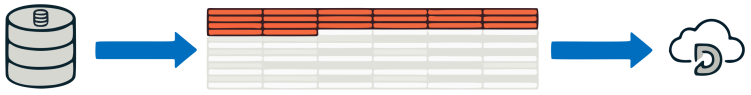
This approach works well when:

- Source systems don't support change tracking
- Data volumes are small enough to reprocess efficiently
- You need to rebuild destination tables from scratch periodically
- Data relationships are complex and incremental logic would be error-prone

Full extraction becomes impractical as data volumes grow. Reprocessing millions of records when only a few thousand changed wastes compute resources and delays downstream consumers.

Incremental extraction

Incremental extraction processes only new or changed records since the last ingestion run. The engine tracks what data has been processed and queries only for updates.



This pattern requires:

- A reliable **change indicator** in the source data (timestamp, sequence number, or version column)
- **State management** to track the last processed position
- Logic to handle **late-arriving** or **out-of-order** records

Incremental extraction significantly improves efficiency for large datasets. Instead of reading millions of rows, you might process only thousands of changes per run.

Streaming extraction

Streaming extraction provides **near real-time** data capture through continuous processing. Rather than running on a schedule, streaming jobs remain active and process records as they arrive.

Consider streaming when:

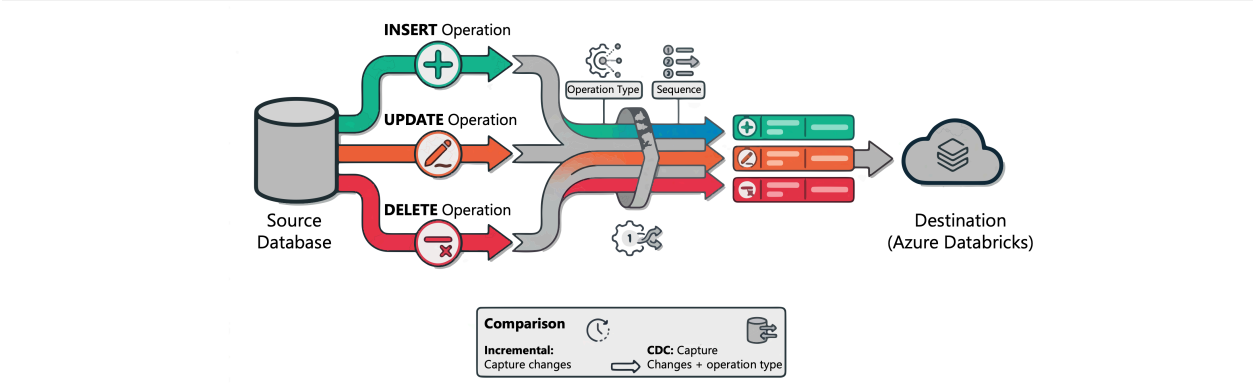
- Business processes require data freshness measured in seconds or minutes
- Source systems publish events to message buses or change feeds

- You need to react to events as they occur

Streaming extraction requires always-on infrastructure but delivers the lowest latency for data availability.

Design for change data capture

Change data capture (CDC) is a specialized extraction pattern that captures individual row-level changes, including inserts, updates, and deletes. CDC differs from simple incremental extraction because it preserves the operation type for each change.



When designing for CDC, consider how your source system produces change records:

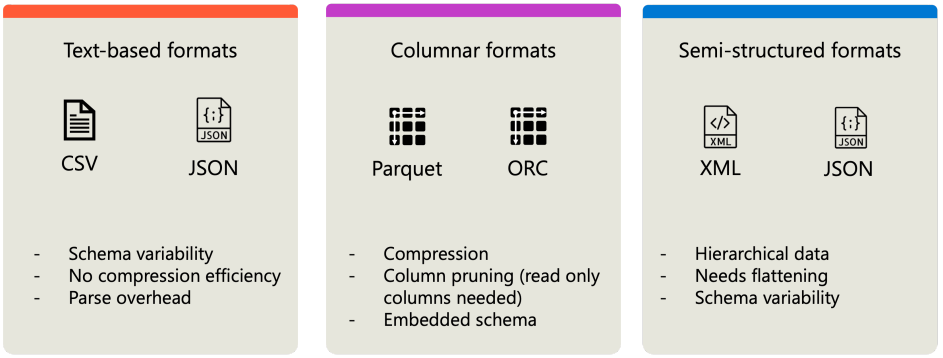
- **Database transaction logs**: Databases like SQL Server can expose row-level changes using its transaction log–based CDC feature. Downstream systems such as Azure Databricks can then read these changes and process them as part of the ingestion pipeline.
- **Change data feeds**: Delta tables and some databases like Azure Cosmos DB provide built-in change tracking that surfaces row changes with metadata.
- **Periodic snapshots**: Some systems don't support continuous CDC, requiring you to compare snapshots to determine changes.

CDC enables use cases that simple incremental extraction can't support. You can accurately replicate deletes, maintain audit trails, and process updates that modify key columns.

> **Note**
>
> CDC requires sequencing information to handle out-of-order events correctly. Your source data needs a column (such as a timestamp or sequence number) that establishes the correct order of changes.

Identify source file type characteristics

When data arrives as files rather than from transactional systems, the file format influences your ingestion design. Each format has distinct characteristics that affect processing.



Text-based formats

CSV and **JSON** files are common source formats because they're human-readable and widely supported. However, they present challenges for efficient processing:

- **Schema variability**: Files might have inconsistent column orders, missing fields, or data type variations
- **No compression efficiency**: Text formats typically consume more storage than binary alternatives
- **Parse overhead**: Every value requires text-to-type conversion during reads

Use text-based formats when source systems can only produce these formats or when data volumes are small enough that efficiency concerns don't apply.

Columnar formats

Parquet and **ORC** files store data in columnar format, enabling efficient analytical queries. These formats offer:

- **Compression**: Columnar storage achieves better compression ratios by grouping similar values
- **Column pruning**: Queries read only the columns they need, reducing I/O
- **Embedded schema**: The file contains metadata describing column names and types

Design your ingestion to favor columnar source formats when you have control over how upstream systems export data.

Semi-structured formats

XML and nested JSON files represent hierarchical data that doesn't fit a flat tabular structure. When ingesting semi-structured data:

- Plan for schema extraction or definition.
- Determine whether to flatten the structure or preserve nesting.

- Account for varying document structures within the same source.

Data source connection considerations

Data source configuration defines how Azure Databricks connects to and authenticates with source systems. Your design must address connection security, credential management, and access patterns.

Cloud object storage sources

Cloud storage (Azure Data Lake Storage, Amazon S3, or Google Cloud Storage) serves as the most common landing zone for ingestion. When configuring storage connections:

- Use Unity Catalog external locations to govern access to storage paths.
- Configure appropriate authentication methods (managed identity, service principals, or storage credentials).
- Design folder structures that support efficient incremental processing.

Folder structure significantly impacts how efficiently Azure Databricks can identify and process new files. Consider these patterns:

Date-based partitioning: Organize files by date hierarchy to enable partition pruning during incremental reads:

```
/data/sales/year=2025/month=11/day=28/
/data/sales/year=2025/month=11/day=29/
```

This structure allows queries to scan only relevant date partitions rather than listing all files.

Source-based organization: When ingesting from multiple sources, separate data by source system to simplify processing logic:

```
/landing/erp/orders/
/landing/crm/customers/
/landing/web/clickstream/
```

Processing state folders: Use separate folders to track file processing status:

```
/incoming/     # New files arrive here
/processing/   # Files currently being processed
/archive/      # Successfully processed files
/failed/       # Files that failed processing
```

When designing folder structures, consider:

- **File discovery overhead**: Flat structures with thousands of files slow down listing operations. Partitioned structures reduce the files Azure Databricks must enumerate.

- **Auto Loader compatibility**: Azure Databricks **Auto Loader** efficiently tracks new files in cloud storage. Structure folders to align with how Auto Loader discovers and checkpoints files.
- **Partition column extraction**: Use Hive-style partitioning (`key=value`) so Azure Databricks can automatically extract partition values as columns without parsing file contents.

Database sources

Connecting to transactional databases requires different considerations than cloud storage:

- **Network connectivity** between Azure Databricks and the database server.
- Authentication credentials stored securely (using **secrets** rather than hardcoded values).
- **Connection pooling** and **query pushdown** capabilities.
- Impact on source system performance during extraction.

Streaming sources

Message buses and event streams (such as Azure Event Hubs or other common messaging platforms) require:

- Broker connection details and authentication.
- **Consumer group** configuration for parallel processing.
- **Offset management** to track consumption position.
- **Schema registry** integration for message format validation.

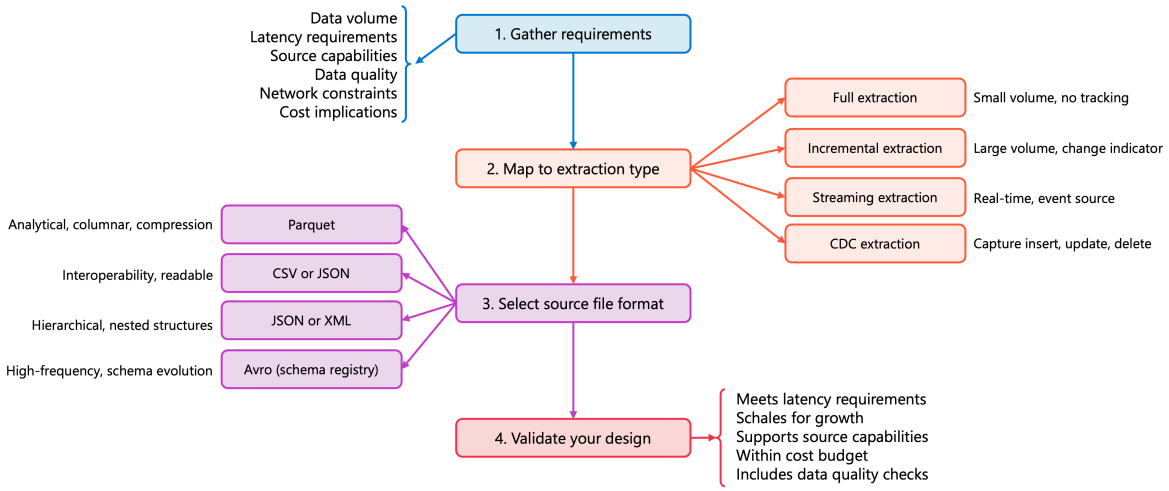
SaaS application sources

Enterprise applications like Salesforce, ServiceNow, or SAP expose data through APIs with unique characteristics:

- API rate limits that constrain extraction throughput.
- Pagination strategies for large result sets.
- Authentication flows (OAuth, API keys, or certificate-based).
- Available change tracking or webhook capabilities.

Apply a requirements-driven design framework

Effective ingestion design starts with analyzing requirements before selecting extraction types or file formats. Use the following repeatable framework to guide your design decisions.



Step 1: Gather requirements

Before making any technical decisions, answer these key questions:

Requirement Category	Questions to Answer
Data volume	How much data exists in the source? How much changes between runs? What's the growth rate?
Latency requirements	How fresh must data be for downstream consumers? Are there SLAs to meet?
Source capabilities	Does the source support change tracking, CDC, or only full extraction? What file formats can it produce?
Data quality	How reliable is the source data? What validation or cleansing is needed?
Network constraints	What bandwidth exists between source and Azure Databricks? Are there firewall or VPN requirements?
Cost implications	What are the compute, storage, and data transfer costs of different approaches?

Step 2: Map requirements to extraction type

Use your requirements analysis to select the appropriate extraction pattern:

If your requirements indicate...	Then choose...
Small data volume (< 100K rows), no change tracking available, simple rebuild acceptable	**Full extraction**
Large data volume, reliable change indicator exists (timestamp/sequence), hourly or daily freshness sufficient	**Incremental extraction**
Real-time or near real-time freshness required, source publishes events, streaming infrastructure available	**Streaming extraction**

If your requirements indicate...	Then choose...
Need to capture deletes and operation types, audit trail required, source supports transaction log capture	**CDC extraction**

Step 3: Select source file format

When you control the source format or need to choose a landing zone format, match the format to your needs:

If your scenario involves...	Then prefer...
Large analytical datasets, columnar queries, need compression	**Parquet**
Interoperability with many systems, human readability needed	**CSV or JSON**
Hierarchical or nested data structures	**JSON or XML**
High-frequency streaming with schema evolution	**Avro with schema registry**

Step 4: Validate your design

Before implementing, verify your design addresses these concerns:

- **The extraction type meets latency requirements.** If not, downstream consumers receive stale data, potentially causing incorrect business decisions or SLA violations.
- **The approach scales to handle projected data growth.** If not, pipelines that work today may fail or become prohibitively slow as data volumes increase.
- **Source system capabilities support your chosen pattern.** If not, the pipeline fails at runtime or requires costly workarounds that add complexity.
- **Cost estimates are within budget constraints.** If not, the solution may be technically sound but financially unsustainable for production use.
- **Data quality checks are planned for the pipeline.** If not, invalid or corrupted data propagates to downstream systems, causing errors or incorrect analytics.

Apply the framework: A practical example

An e-commerce platform needs to ingest order data. Orders are stored in a PostgreSQL database, and the business requires hourly reporting.

Step 1 - Gather requirements:

- **Volume**: 50,000 new orders daily, 2 million total rows, 15% annual growth.
- **Latency**: One-hour freshness acceptable for reporting.
- **Source**: PostgreSQL supports timestamp-based queries but no native CDC.
- **Quality**: Source data is validated at entry; minimal cleansing needed.

- **Network**: Direct connectivity via Azure Private Link.
- **Cost**: Minimize compute usage; storage costs are secondary.

Step 2 - Map to extraction type:

Large volume with reliable timestamp column and hourly latency requirement → **Incremental extraction**.

Why not the others?

- **Full extraction**: Reprocessing 2 million rows hourly wastes compute when only ~2,000 orders change per hour.
- **Streaming**: One-hour latency doesn't justify the cost of always-on streaming infrastructure.
- **CDC**: PostgreSQL doesn't expose native transaction log CDC, and the use case doesn't require tracking deletes or operation types.

Step 3 - Select file format:

Data lands in Azure Data Lake Storage for processing → **Parquet** for efficient columnar storage. Date-partitioned layout for incremental discovery → `/landing/orders/year=2025/month=11/day=28/`.

Why not the others?

- **CSV/JSON**: Larger file sizes, no compression benefits, and slower query performance for analytical workloads.
- **Avro**: Better suited for streaming with schema evolution; batch ingestion doesn't need its streaming-oriented features.
- **XML**: Adds parsing complexity without benefits for structured tabular order data.

Step 4 - Validate design:

- ✓ Hourly incremental meets one-hour latency requirement.
- ✓ Processing only changes scales efficiently with growth.
- ✓ PostgreSQL `order_updated_at` column enables incremental queries.
- ✓ Processing thousands vs. millions of rows reduces compute costs.

Final decision: Incremental extraction using the `order_updated_at` timestamp column, scheduled hourly, landing data as Parquet files in Azure Data Lake Storage.

> **Tip**
>
> Document your design decisions and the rationale behind them. When requirements change or issues arise, this documentation helps you understand why the current approach was chosen and what tradeoffs were made. Consider creating a design decision record for each major pipeline.

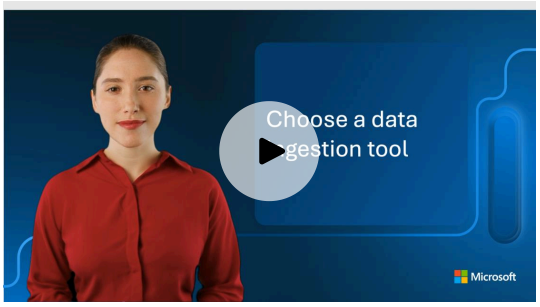
Building on this requirements-driven foundation, you're ready to evaluate the specific tools and loading methods that implement your ingestion design.

3. Choose a data ingestion tool

https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/3-choose-data-ingestion-tool

Choose a data ingestion tool

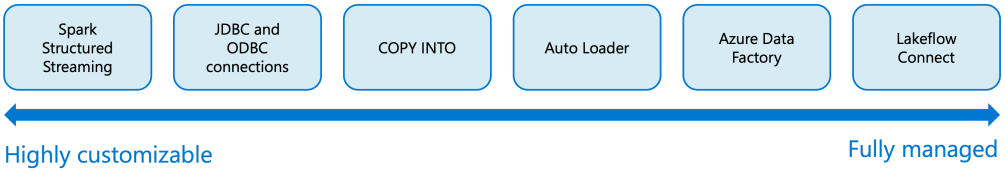
Completed



When you design data pipelines in Azure Databricks, selecting the right ingestion tool directly impacts pipeline reliability, maintenance effort, and long-term costs. Each tool addresses different scenarios, data sources, and operational requirements. Understanding these differences helps you make informed decisions that align with your organization's needs.

Understand the ingestion tool landscape

Azure Databricks provides multiple ingestion options that span a spectrum from fully managed services to highly customizable frameworks. The tools you choose depend on factors such as your data source type, latency requirements, and how much control you need over the ingestion process.



At the most managed end, **Lakeflow Connect** offers turnkey connectors for enterprise applications and databases. These connectors handle authentication, change tracking, error recovery, and schema evolution automatically. At the customizable end, **Spark Structured Streaming** gives you complete control over how data flows through your pipeline, though it requires more development and maintenance effort.

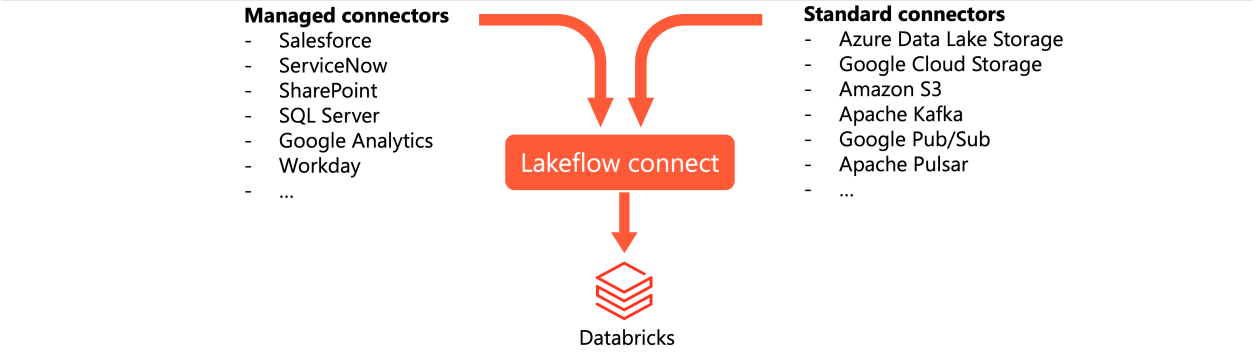
Between these extremes, you find tools like **Auto Loader** for cloud storage files, **COPY INTO** for SQL-based batch loading, and notebook-based approaches using **JDBC/ODBC** connections. Additionally, **Azure Data Factory** serves as an orchestration layer that can coordinate data movement into Azure Databricks from the broader Azure ecosystem.

Lakeflow Connect

Lakeflow Connect represents the modern approach to data ingestion in Azure Databricks. It offers both fully managed connectors and standard connectors that integrate with Lakeflow Spark Declarative Pipelines for a streamlined experience.

Managed connectors provide out-of-the-box integration with popular enterprise applications and databases. Supported sources include Salesforce, ServiceNow, SharePoint, SQL Server, Google Analytics, and Workday. These connectors handle the complexities of source-specific authentication, incremental data capture, API rate limiting, and automated retries. They run on serverless compute and write to streaming tables governed by Unity Catalog.

Standard connectors extend Lakeflow Connect to cloud object storage and message buses. You can ingest data from Amazon S3, Azure Data Lake Storage, or Google Cloud Storage using Auto Loader. For real-time data, standard connectors support Apache Kafka, Google Pub/Sub, and Apache Pulsar.



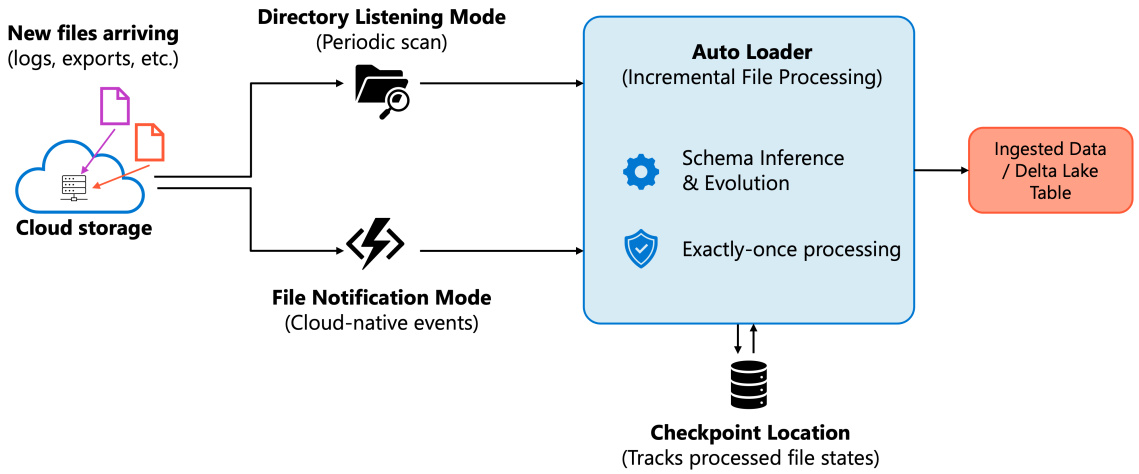
Consider Lakeflow Connect when you need reliable, low-maintenance ingestion from supported sources. The managed connectors significantly reduce development time and ongoing maintenance compared to custom solutions. However, if your data source isn't supported or you need highly specialized ingestion logic, you might need to explore other options.

Lakeflow Connect supports both **scheduled** ingestion (hourly, daily, weekly) and **on-demand** triggers, making it suitable for regular data refresh cycles and ad hoc loading scenarios.

Auto Loader

Auto Loader provides incremental file processing for data arriving in cloud storage. Rather than scanning entire directories to find new files, Auto Loader efficiently tracks which files have been processed and ingests only new arrivals.

Auto Loader works by maintaining state in a checkpoint location that tracks processed files. When new files land in your cloud storage, Auto Loader discovers them through either directory listing or file notification mode. Directory listing periodically scans for new files, while file notification mode uses cloud-native events for immediate detection.



Key capabilities of Auto Loader include:

- **Schema inference and evolution**: Automatically detects the schema of incoming files and handles schema changes gracefully.
- **Exactly-once processing**: Guarantees each file is processed once, even across failures and restarts.
- **Scalability**: Efficiently handles billions of files for backfills or migrations.
- **Format support**: Processes JSON, CSV, Parquet, Avro, ORC, XML, text, and binary files.

Auto Loader fits scenarios where data arrives as files in cloud storage—common patterns include log files, exports from upstream systems, and data landing zones. You can use Auto Loader with Lakeflow Spark Declarative Pipelines for a managed experience or with Structured Streaming for more control.

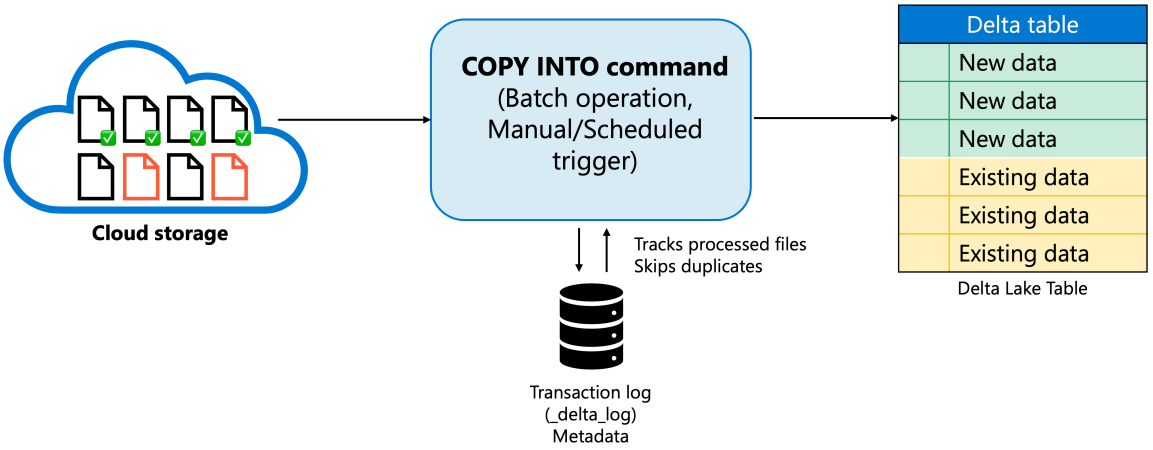
COPY INTO

The `COPY INTO` SQL command loads data from cloud storage into Delta tables. Unlike Auto Loader, which runs continuously as a stream, `COPY INTO` executes as a batch operation that you schedule or trigger manually.

`COPY INTO` tracks which files it has already processed, making the operation idempotent. Running the same command multiple times doesn't duplicate data because previously loaded files are skipped. This characteristic makes `COPY INTO` useful for scheduled batch jobs where you want predictable, repeatable behavior.

> **Note**

> The tracking metadata is stored as part of the Delta table's transaction log (the `_delta_log` directory)



Common scenarios for `COPY INTO` include one-time data migrations, scheduled batch loads from landing zones, and SQL-first workflows where you prefer declarative syntax over streaming APIs.

Choose `COPY INTO` over Auto Loader when you're performing a one-time batch load of a large dataset. `COPY INTO` provides a simpler configuration model without requiring streaming infrastructure, making it ideal for loading hundreds of gigabytes or even terabytes of data in a single operation. Auto Loader's continuous processing model and checkpoint management add unnecessary complexity when you only need to load data once.

However, for ongoing incremental ingestion at scale, Auto Loader provides better performance and more robust file discovery.

Spark Structured Streaming

Spark Structured Streaming provides the foundation for real-time data processing in Azure Databricks. It enables continuous ingestion from various streaming sources with exactly-once processing guarantees.

Streaming sources supported include:

- **Apache Kafka and Azure Event Hubs**: For event-driven architectures and message queue processing.
- **Delta Lake change feeds**: For propagating changes between Delta tables.
- **Cloud storage via Auto Loader**: For file-based streaming ingestion.
- **Socket and custom sources**: For specialized ingestion scenarios.

Choose Structured Streaming when you need fine-grained control over streaming behavior, custom watermarking or windowing logic, or when working with sources not supported by higher-level abstractions.

JDBC and ODBC connections

Notebook-based ingestion using JDBC or ODBC remains a practical approach for extracting data from relational databases. You write code that connects directly to source databases, executes queries, and loads results into Delta tables.

This approach supports virtually any database with a JDBC or ODBC driver: SQL Server, Oracle, PostgreSQL, MySQL, and many others. You implement full or incremental loads by writing appropriate queries, often using watermark columns to track extraction progress.

While Lakeflow Connect managed connectors are replacing JDBC/ODBC for many scenarios, notebook-based connections remain valuable when:

- Your database isn't supported by Lakeflow Connect managed connectors.
- You need custom extraction logic that managed connectors can't accommodate.
- You're maintaining existing pipelines that use this pattern.
- You require specific query pushdown optimizations for large tables.

When using JDBC/ODBC, store credentials securely using Databricks secrets rather than hardcoding them in notebooks. Consider the performance impact on source systems and implement appropriate throttling or scheduling to avoid overwhelming transactional databases.

For on-premises databases, additional network infrastructure is required—either Azure ExpressRoute/VPN for direct JDBC connectivity from Databricks, or Azure Data Factory with a Self-Hosted Integration Runtime to land data in cloud storage first.

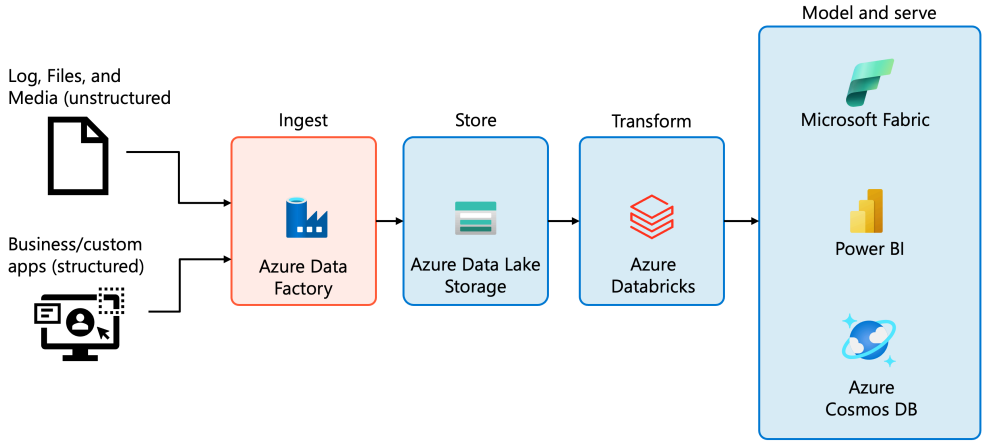
Azure Data Factory

Azure Data Factory (ADF) serves as an orchestration and data movement platform within the Azure ecosystem. It excels at coordinating workflows that span multiple services and landing data from diverse sources into cloud storage where Azure Databricks can process it.

ADF integrates extensively with Microsoft services like Dynamics 365, Azure SQL Database, and Microsoft Fabric. When your data originates from these sources, ADF provides straightforward connectivity and can land data in Azure Data Lake Storage for subsequent Databricks processing.

Understanding ADF's appropriate role is important. Use ADF for:

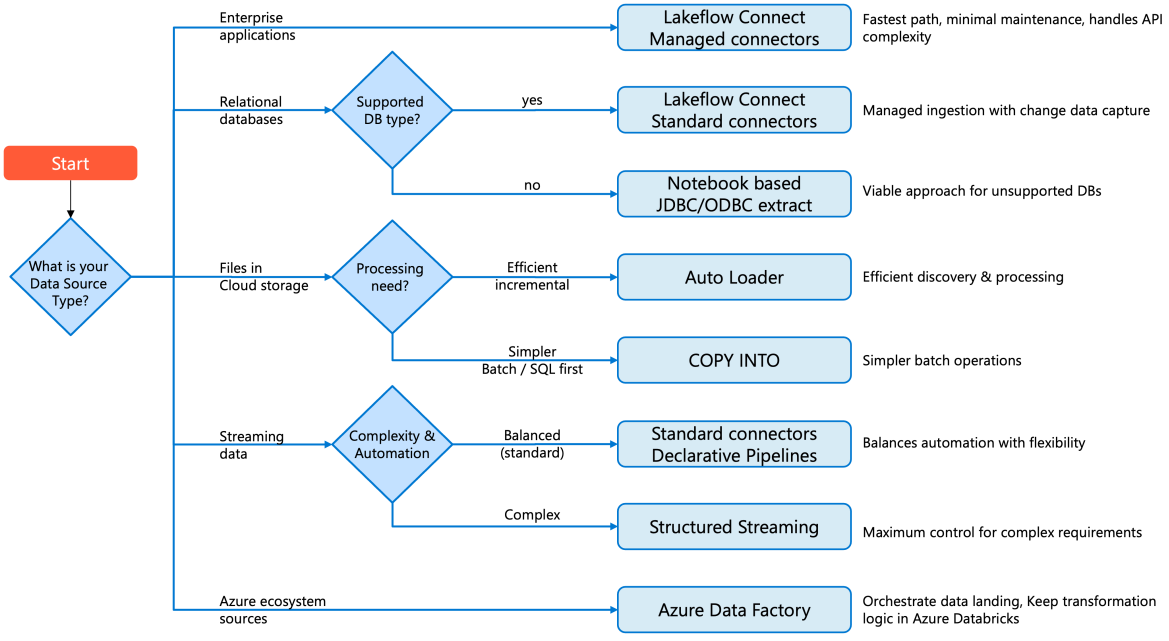
- **Orchestration**: Coordinating workflows that involve multiple Azure services.
- **Initial data landing**: Moving data from Azure sources to cloud storage.
- **Enterprise integration**: When company policy mandates ADF for data movement.



Avoid using ADF for core ETL transformations inside Databricks. Running transformations through ADF activities introduces unnecessary complexity and performs less efficiently than Databricks-native processing. Let ADF handle data movement into your lakehouse, then use Azure Databricks tools for transformation logic.

Apply a decision framework

Selecting an ingestion tool becomes clearer when you evaluate your scenario against key decision criteria.



Start by identifying your data source type, then consider operational requirements and organizational context.

For **enterprise applications** (Salesforce, ServiceNow, SharePoint), Lakeflow Connect managed connectors offer the fastest path to production with minimal maintenance. These connectors handle the complexity of API integration, rate limiting, and change tracking.

For **relational databases** with supported types, Lakeflow Connect database connectors provide managed ingestion with change data capture. For unsupported databases, notebook-based JDBC/ODBC extraction remains a viable approach.

For **files in cloud storage**, Auto Loader provides efficient incremental discovery and processing. Use COPY INTO for simpler batch scenarios or when you prefer SQL-first workflows.

For **streaming data** from message buses (Kafka, Event Hubs, Pub/Sub), standard connectors with Lakeflow Spark Declarative Pipelines balance automation with flexibility. For complex streaming requirements, Structured Streaming offers maximum control.

For **Azure ecosystem sources** requiring coordination with other services, Azure Data Factory can orchestrate data landing into your lakehouse. Keep transformation logic within Azure Databricks rather than in ADF data flows.

> **Tip**
>
> Start with the most managed option that meets your requirements. Managed connectors reduce development time and maintenance burden. Only move to more customizable approaches when specific requirements can't be met by higher-level abstractions.

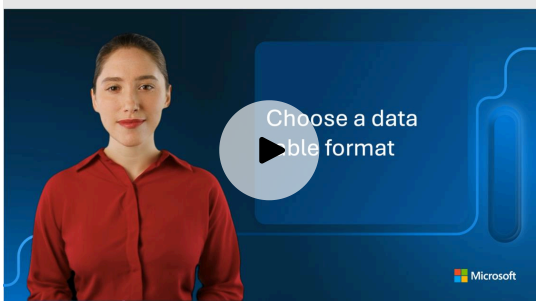
Your choice also depends on team skills and organizational standards. Document your decisions and the reasoning behind them to help future team members understand the pipeline architecture.

4. Choose a data table format

https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/4-choose-data-table-format

Choose a data table format

Completed

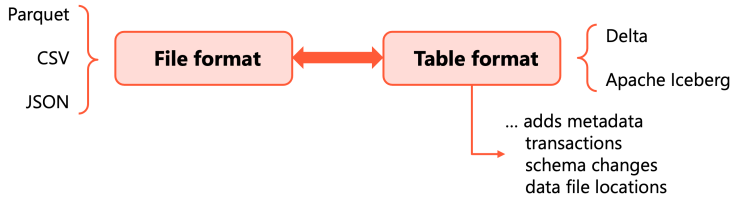


Data engineers face a critical decision early in every data pipeline: which table format should store your data? This choice affects query performance, transaction reliability, interoperability with other platforms, and long-term maintenance of your lakehouse architecture.

Azure Databricks supports multiple storage formats, each designed for different scenarios. Understanding their differences helps you select the right format for your specific requirements.

Understand file formats versus table formats

Before comparing options, it's important to distinguish between file formats and table formats. A **file format** like Parquet, CSV, or JSON defines how data is physically stored on disk. A **table format** like Delta Lake or Apache Iceberg builds on top of file formats by adding a metadata layer that tracks transactions, schema changes, and data file locations.



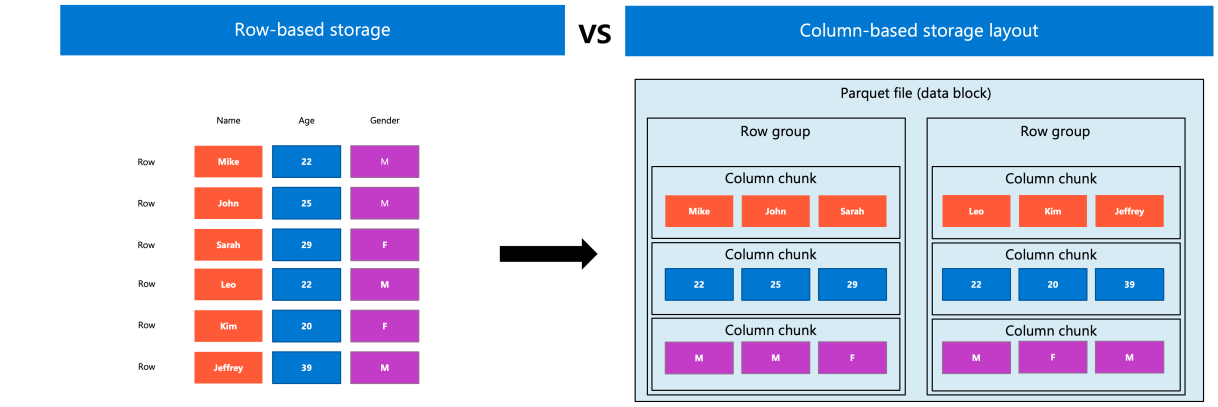
Both Delta Lake and Apache Iceberg use Parquet as their underlying file format. The key difference lies in the metadata layer that enables advanced features like ACID transactions, time travel, and schema evolution.

How Parquet stores data differently

Parquet uses **columnar storage** rather than row-based storage. This fundamental difference significantly impacts analytical query performance.

In row-based formats like CSV, data is organized by record. Reading a single column requires scanning every row in the file. In columnar formats like Parquet, data for each column is stored together. When a query needs only specific columns, Parquet reads just those columns and skips the rest entirely.

Parquet organizes data into **row groups**, each containing **column chunks**. Every column chunk stores statistics including minimum and maximum values for that data segment. When you execute a query with a filter condition, the query engine checks these statistics first. If the filter value falls outside the min/max range of a row group, the engine skips reading that entire section.



This columnar approach provides two optimization techniques that dramatically improve analytics performance:

- **Column pruning**: The engine reads only the columns your query references, reducing I/O substantially.
- **Predicate pushdown**: Filter conditions are evaluated against column statistics before reading data, skipping irrelevant row groups entirely.

For analytical workloads that typically access a subset of columns across many rows, Parquet substantially outperforms row-based formats.

Delta Lake as the default format

Delta Lake is the default storage format for all tables in Azure Databricks. When you create a table without specifying a format, Azure Databricks automatically uses Delta Lake.

Delta Lake extends Parquet files with a **transaction log** that records every change to your table. This log enables several capabilities that Parquet alone doesn't provide:

- **ACID transactions**: Multiple concurrent readers and writers can access the same table without data corruption. Each transaction either completes entirely or has no effect.
- **Time travel**: Query previous versions of your table using timestamps or version numbers. Roll back changes if needed.
- **Schema enforcement and evolution**: The table validates incoming data against its defined schema, preventing corrupt or incompatible data from entering your lakehouse. When requirements change, you can add new columns, modify data types, or rename columns without rewriting existing data.
- **Unified batch and streaming**: Use the same table as both a batch source and a streaming source or sink.

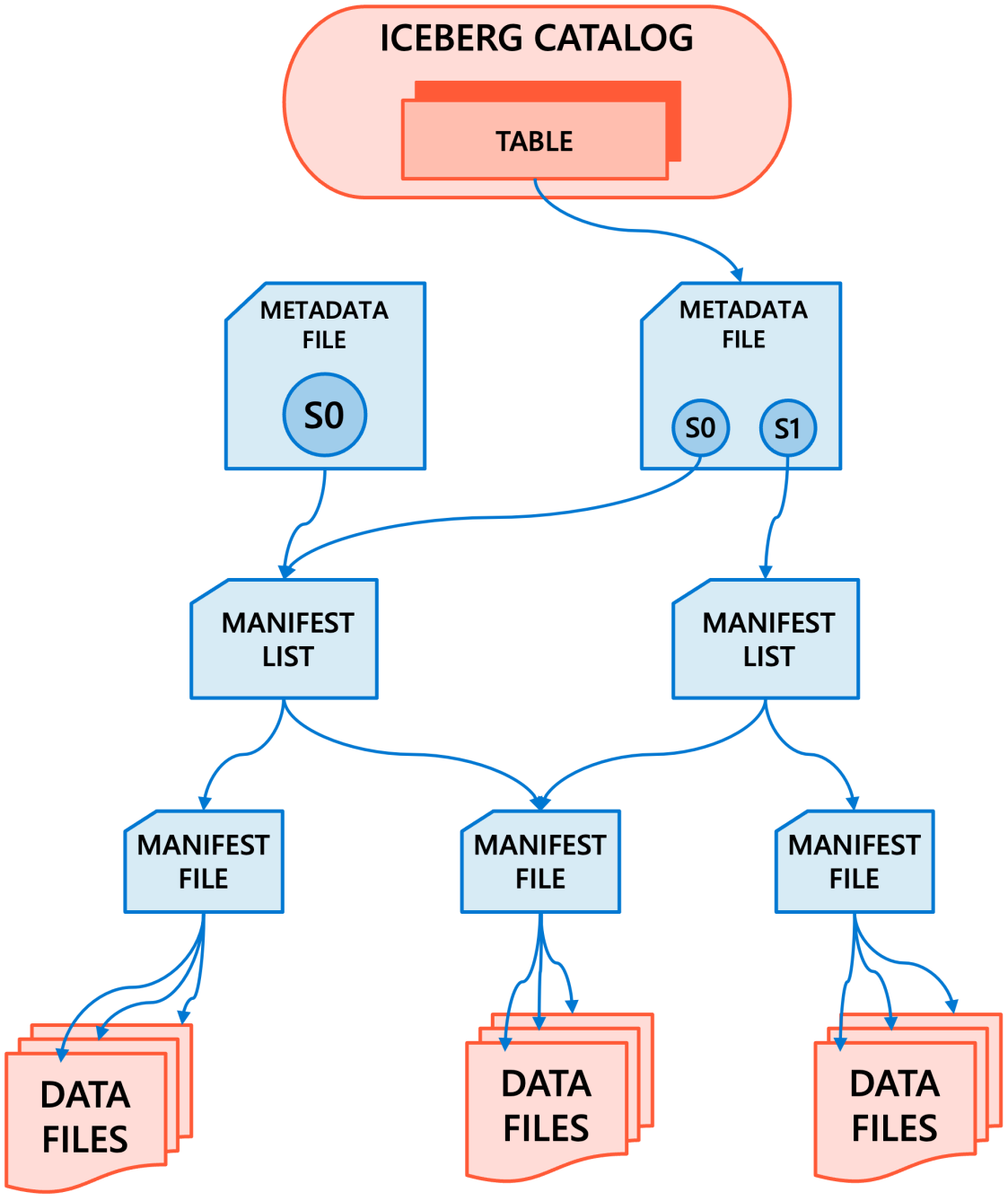
Delta Lake integrates deeply with the Azure Databricks platform. Features like **liquid clustering**, **predictive optimization**, and **change data feed** rely on Delta Lake's transaction log.

For most scenarios in Azure Databricks, Delta Lake is the recommended choice. It provides the performance benefits of Parquet with transactional guarantees that production data systems require.

Apache Iceberg for cross-platform scenarios

Apache Iceberg is an alternative open table format that Azure Databricks supports for managed and foreign tables. Like Delta Lake, Iceberg provides ACID transactions, schema evolution, and time travel capabilities built on top of Parquet files.

Iceberg's hierarchical and well-defined logical structure makes it highly flexible, supporting scalability and advanced optimizations. It supports manifest pruning, file-Level statistics, hidden partitioning and partition evolution.



The primary reason to choose Iceberg is **interoperability**. Iceberg is widely adopted across the data ecosystem, including platforms like Snowflake, Amazon Athena, Trino, Apache Spark, and Apache Flink. If your organization runs workloads across multiple platforms that need to read and write the same tables, Iceberg provides a common format all platforms understand.

Unity Catalog can manage Iceberg tables directly, allowing Azure Databricks to serve as an Iceberg catalog for external engines. You can also read Iceberg tables managed by other catalogs like AWS Glue or Snowflake Horizon Catalog through foreign table connections.

Consider Iceberg when:

- Your data needs to be accessed by multiple processing engines outside Azure Databricks.
- Your organization has standardized on Iceberg for cross-platform data sharing.
- You're integrating with systems that don't support Delta Lake.

For workloads that run primarily within Azure Databricks, Delta Lake remains the recommended format due to its deeper platform integration and optimization features.

When row-based formats apply

CSV and JSON remain valid choices for specific scenarios, even though they're less efficient for large-scale analytics.

Use CSV or JSON when:

- You're receiving data from external systems in these formats as an initial landing zone.
- You need human-readable data for debugging or quick inspection.
- You're integrating with systems that only support text-based formats.
- Data volumes are small enough that performance differences are negligible.

For persistent storage of analytical data, convert incoming CSV or JSON files to **Delta Lake tables** during ingestion. This conversion captures the efficiency gains of columnar storage while adding transactional guarantees.

Apply decision criteria

Selecting a table format depends on your specific requirements. Start with Delta Lake as your default choice, then evaluate alternatives only when specific constraints require them.

Consideration	Delta Lake	Apache Iceberg	CSV/JSON
Analytics performance	Excellent	Excellent	Poor
ACID transactions	Yes	Yes	No
Time travel	Yes	Yes	No
Azure Databricks integration	Deep	Supported	Basic
Cross-platform compatibility	Growing	Wide	Universal
Recommended for production	Yes	Yes (when interop required)	No (landing zone only)

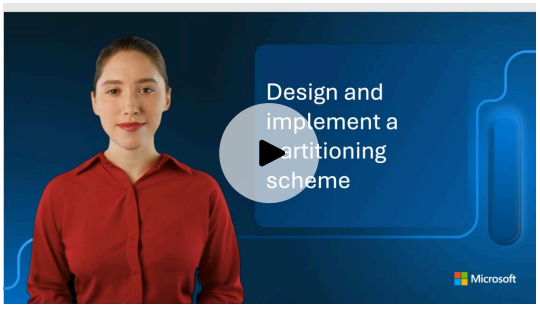
Your choice also depends on organizational standards. If your architecture team has established conventions for table formats, align with those decisions to maintain consistency across your data platform.

5. Design and implement a data partitioning scheme

https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/5-design-data-partitioning-scheme

Design and implement a data partitioning scheme

Completed



Data partitioning directly affects how efficiently your queries run and how much compute resources you consume. When you partition a table, you divide data into discrete segments based on column values, allowing the query engine to read only relevant portions of your dataset. This approach, called **partition pruning**, can dramatically reduce data scans when filters align with partition boundaries.

Consider a retail company processing millions of daily sales transactions. Analysts frequently query data by date ranges to generate weekly reports, compare month-over-month trends, and analyze seasonal patterns. Without partitioning, every query scans the entire transaction history. With proper partitioning by date, queries for a specific week touch only a fraction of the stored files.

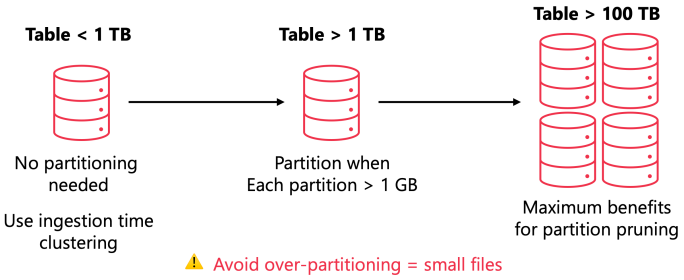
Determine when partitioning applies

Databricks recommends that you don't partition tables containing less than **1 TB** of data. For smaller tables, the overhead of managing partition metadata and the potential for creating many small files outweighs any query performance benefits. Unpartitioned Delta tables in recent Databricks Runtime versions automatically benefit from **ingestion time clustering**, which provides similar query advantages without explicit configuration.

When your tables exceed **1 TB**, partitioning becomes a valuable optimization technique. At this scale, the benefits of reading only relevant data segments significantly offset the costs of partition management. Partitioning also enables parallel processing, as Spark can distribute different partitions across cluster nodes for concurrent execution.

> **Important**

> Each partition should contain at least **1 GB** of data. This guideline helps you avoid over-partitioning, which creates excessive small files and degrades query performance.



For very large tables in the hundreds of terabytes range, partitioning delivers even greater benefits. At this scale, partition pruning dramatically reduces the amount of data the query engine needs to process, improving both performance and cost efficiency.

Choose effective partition keys

Partition key selection requires you to understand your data characteristics and query patterns. Effective partition keys share common traits that maximize performance benefits while avoiding common pitfalls.

Low cardinality columns work best

Choose columns with a **limited number of distinct values**. Date columns, geographic regions, product categories, and status fields make excellent partition keys because they create a manageable number of partitions. A column with thousands or millions of unique values, like customer ID or transaction ID, leads to over-partitioning and poor performance.

For example, partitioning a sales table by `sale_date` at the daily level might create 365 partitions per year. If each day contains sufficient data volume, this granularity works well. However, partitioning by `sale_timestamp` at the second level would create millions of partitions, each containing minimal data.

Align with query patterns

Your partition key should match the columns that users filter on most frequently. If analysts consistently query sales data by date ranges, partitioning by date enables efficient pruning. If queries primarily filter by region, consider region as your partition key instead.

When multiple filter patterns exist, prioritize the most common or most performance-critical queries. You can define one partitioning scheme per table, so choose columns that benefit the majority of your workload. For significantly different query patterns, consider creating separate optimized **materialized views** or **summary tables** for specific use cases.

Consider data growth over time

Select a partition scheme that scales appropriately as your data grows. Partitioning by year works well initially but might create partitions that are too large over time. Conversely, partitioning by hour might work for high-volume data but creates too many small partitions for moderate volumes.

Evaluate your expected data growth rate when choosing partition granularity. A balance between partition size and the number of partitions helps maintain consistent query performance as your dataset expands.

Also consider data distribution across partitions. Severely uneven partition sizes can create processing bottlenecks, as nodes handling larger partitions take longer to complete while others remain idle.

Implement partitions in Azure Databricks

You define partitions when creating a table using the `PARTITIONED BY` clause in SQL or the `partitionBy` method in PySpark. Once you establish partitions, Azure Databricks automatically routes incoming data to the appropriate partition directories based on column values.

Create a partitioned table with SQL

The following example creates a sales transactions table partitioned by the transaction date:

```
CREATE TABLE sales.transactions (
    transaction_id STRING,
    customer_id STRING,
    product_id STRING,
    quantity INT,
    unit_price DECIMAL(10,2),
    total_amount DECIMAL(12,2),
    transaction_date DATE
)
PARTITIONED BY (transaction_date);
```

When you query this table with a date filter, the query engine reads only the partitions that match your filter criteria:

```
SELECT product_id, SUM(total_amount) AS revenue
FROM sales.transactions
WHERE transaction_date BETWEEN '2024-01-01' AND '2024-01-31'
GROUP BY product_id;
```

Create a partitioned table with PySpark

When using PySpark, specify partitioning during the write operation:

```
df.write.format("delta") \
    .partitionBy("transaction_date") \
    .saveAsTable("sales.transactions")
```

For subsequent writes, maintain consistency by including the same partition specification:

```
new_transactions_df.write.format("delta") \
    .mode("append") \
    .partitionBy("transaction_date") \
    .saveAsTable("sales.transactions")
```

Use generated columns for partition flexibility

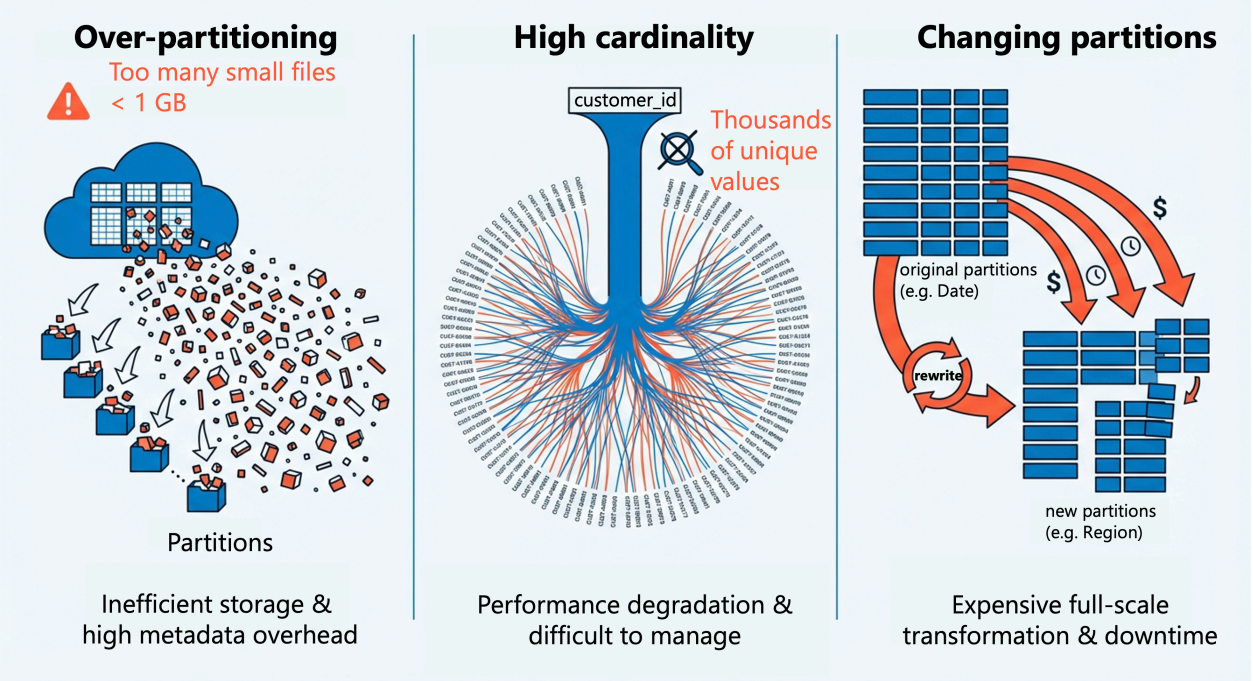
Generated columns provide a way to derive partition values from existing columns without storing redundant data. This approach is useful when your source data contains timestamps but you want to partition by date:

```
CREATE TABLE sales.transactions (
    transaction_id STRING,
    customer_id STRING,
    product_id STRING,
    quantity INT,
    unit_price DECIMAL(10,2),
    total_amount DECIMAL(12,2),
    transaction_timestamp TIMESTAMP,
    transaction_date DATE GENERATED ALWAYS AS (CAST(transaction_timestamp AS DATE))
)
PARTITIONED BY (transaction_date);
```

> **Tip**
>
> With this configuration, Delta Lake automatically generates partition filters when you query using the base timestamp column. A query filtering on `transaction_timestamp` benefits from partition pruning without requiring explicit date filters.

Avoid common partitioning mistakes

Understanding what to avoid is as important as knowing what to do. Several common mistakes can significantly degrade performance and complicate data management.



Over-partitioning creates small file problems

Creating too many partitions, each containing insufficient data, leads to **small file problems**. When partitions contain files smaller than the target file size, query performance suffers from excessive metadata operations and reduced read efficiency. Aim for partitions with at least **1 GB** of data.

Signs of over-partitioning include slow query performance despite partition pruning, a large number of small files in partition directories, and long table optimization times. If you observe these symptoms, consider choosing a coarser partition granularity.

Partitioning on high-cardinality columns fails

Columns with many unique values make poor partition keys. Partitioning by customer ID, transaction ID, or other **high-cardinality columns** creates millions of partitions that degrade rather than improve performance. Reserve high-cardinality column optimization for other techniques that are designed for this purpose.

Changing partitions requires rewriting data

Partition columns are part of the table structure. Changing your partitioning strategy requires **rewriting the entire table** to a new location with different partition boundaries. This operation consumes significant compute resources and time for large tables.

Plan your partitioning strategy carefully before loading data at scale. Consider future query patterns and data growth to avoid costly restructuring later.

Document partitioning decisions

Record your partition key choices and the reasoning behind them. This documentation helps team members understand query optimization opportunities and prevents accidental changes that could impact performance. Include information about expected partition sizes, query patterns that benefit from the scheme, and any constraints that influenced your decisions.

Consider liquid clustering as an alternative

When you encounter partitioning challenges—especially with **high-cardinality columns** like timestamps or when query patterns are **uncertain**—liquid clustering provides a modern alternative. Unlike traditional partitioning, liquid clustering offers:

- Flexibility to adjust optimization strategies without rewriting data
- Automatic maintenance as tables grow
- Reduced partition management overhead

Liquid clustering is the **recommended approach** for new tables where partitioning constraints would limit performance or create excessive maintenance burden. It's particularly effective for tables that exceed 1 TB and require filtering on columns that would create too many partitions using traditional approaches.

> **Note**
>
> Liquid clustering is not compatible with partitioned tables. Choose one approach based on your requirements when designing new tables.

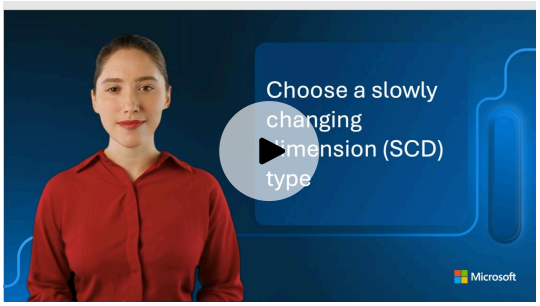
For implementation details, syntax, and optimization strategies, refer to the **Design and implement a clustering strategy** unit later in this module.

6. Choose a slowly changing dimension (SCD) type

https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/6-choose-slowly-changing-dimension-type

Choose a slowly changing dimension (SCD) type

Completed



Dimension tables in a data warehouse capture descriptive attributes about business entities like customers, products, or employees. These attributes change over time, and how you handle those changes directly affects your ability to perform accurate historical analysis. Slowly changing dimensions (SCDs) provide frameworks for managing these changes, and choosing the right SCD type is a critical design decision.

As a data engineer, you're expected to understand SCD types and implement them when directed. While data architects often make the initial design decisions, you need sufficient knowledge to evaluate trade-offs, recommend approaches, and execute implementations correctly. This unit helps you understand the characteristics of each SCD type so you can select the appropriate approach for your business requirements.

Understand dimension changes

Before selecting an SCD type, consider what happens when source data changes. Imagine a customer named Hank Zoeng moves from Seattle to Portland. Your customer dimension table currently shows Seattle as his city. You have three fundamental options:

- **Overwrite** the existing value with Portland, losing all record that he was ever in Seattle.
- **Preserve** the Seattle record and create a new version showing Portland, maintaining the complete history.
- **Track** a limited history by adding a "previous city" column alongside the current value.

Each approach serves different analytical needs. Sales reports that show "current customer location" need different handling than analyses tracking "customer migration patterns" or "sales by customer's region at time of purchase."

SCD Type 0

SCD Type 0 represents the simplest approach: never update dimension attributes after the initial insert. The original values remain permanently, regardless of source system changes.

Customer_ID	Customer_Name	Birth_Date	Registration_Date	Country
1001	Alice Smith	1988-04-12	2015-06-01	USA
1002	Bob Johnson	1979-11-03	2016-09-15	Canada

Customer_ID	Attribute Changed	Old Value	New Value	Action Taken
1001	Country	USA	UK	✗ No update — original value preserved

This type works well for:

- **True constants**: Birth dates, original registration dates, or founding dates that genuinely never change.
- **Audit requirements**: When regulations require preserving the original value exactly as first recorded.
- **Reference data**: Lookup values that represent permanent classifications.

Type 0 is rarely used in isolation because most dimension attributes do change over time. However, individual columns within a dimension table might use Type 0 treatment while other columns use different SCD types.

SCD Type 1

SCD Type 1 overwrites the existing dimension record with new values. When Hank moves from Seattle to Portland, you update his row directly. The dimension table always reflects the current state of the source system.

Before update

ID	Name	City	Email
2001	Hank Miller	Seattle	hank.miller@example.com

After update

ID	Name	City	Email
2001	Hank Miller	Portland	hank.miller@example.com

Metric	Before Update	After Update (Type 1)
Total Purchases While in City	Seattle: $10,000	Portland: $10,000
Historical Accuracy Preserved?	✓ Yes (Seattle)	✗ No (value shifts to Portland)

Type 1 is appropriate when:

- **Historical values don't matter**: Correcting data entry errors, updating email addresses, or refreshing phone numbers.
- **Analysis focuses on current state**: Reports that answer "where do our customers live today?" rather than "where did they live when they made purchases?"
- **Simplicity is prioritized**: Type 1 requires no additional columns or versioning logic.

> **Important**
>
> Type 1 changes affect historical aggregations. If Hank purchased $10,000 of products while in Seattle, those sales now appear attributed to Portland after the update. This behavior might misrepresent historical performance by region.

Consider using Type 1 for supplementary attributes that don't affect analytical accuracy, such as:

- Contact information (email, phone, address corrections).
- Descriptive text fields (product descriptions, customer notes).
- Error corrections in any attribute.

SCD Type 2

SCD Type 2 preserves complete history by inserting new rows for each version of a dimension member. When Hank moves to Portland, the existing Seattle row remains and a new Portland row is created. Both rows share the same natural key (customer ID) but have different surrogate keys.

> **Note**
>
> A natural key is also referred to as a **business key** or **source system key**. It represents the unique identifier from the original data source, such as customer ID, employee number, or product code.

Type 2 dimensions require additional tracking attributes:

Attribute	Purpose
Surrogate key	Unique identifier for each version
Validity start date	When this version became effective
Validity end date	When this version expired (often 12/31/9999 for current)
Current flag	Boolean indicating the active version

The following table shows how Hank Zoeng's customer record appears after moving from Seattle to Portland:

SK	CustomerID	Name	City	ValidFrom	ValidTo	Current
1001	C-555	Hank Zoeng	Seattle	2020-01-15	2024-03-01	FALSE
1002	C-555	Hank Zoeng	Portland	2024-03-01	9999-12-31	TRUE

Type 2 is appropriate when:

- **Historical accuracy matters**: Analyzing sales by the customer's location at the time of purchase.
- **Audit trails are required**: Tracking employee assignments, territory changes, or organizational restructuring over time.
- **Trend analysis is needed**: Understanding how dimension attributes evolved.

Type 2 increases complexity and storage requirements. Each version creates a new row, and fact tables reference specific versions through surrogate keys. Query complexity increases because analysts must understand the versioning model.

> **Tip**
>
> Reserve Type 2 for attributes where historical perspective affects business decisions. Don't apply Type 2 to every changing attribute—the overhead isn't justified for supplementary data like phone numbers.

SCD Type 3

SCD Type 3 tracks limited history by adding columns for previous values. Instead of creating new rows, you add a "previous city" column alongside "current city." This approach tracks only the most recent change, not the complete history.

The following table shows how Hank Zoeng's customer record appears after moving from Seattle to Portland:

CustomerID	Name	CurrentCity	PreviousCity
C-555	Hank Zoeng	Portland	Seattle

Type 3 is appropriate when:

- **Only the previous value matters**: Comparing current versus prior organizational structure.
- **Storage constraints exist**: When Type 2 versioning would create too many rows.
- **Analysis compares before and after**: Product reformulations, price tier changes, or policy updates.

Type 3 is the least common SCD type because:

- It captures only one prior value (or a fixed number of prior values).
- Adding columns for each tracked attribute can widen tables significantly.
- Semantic models can find Type 3 structures difficult to use.

Most scenarios that seem to need Type 3 are better served by Type 2, which provides complete history without schema changes for each attribute.

Compare SCD types

The following table summarizes when to select each SCD type:

Consideration	Type 0	Type 1	Type 2	Type 3
History preserved	Original only	None	Complete	Limited
Storage impact	Minimal	Minimal	High	Moderate
Query complexity	Simple	Simple	Complex	Moderate
Common use cases	Constants, audit requirements	Error corrections, contact updates	Sales attribution, territory tracking	Before/after comparisons
Fact table relationship	Natural key	Natural key	Surrogate key	Natural key

Apply mixed SCD strategies

Production dimension tables often combine multiple SCD types. A customer dimension might handle attributes differently based on their analytical significance:

- **Type 0**: Original registration date (never changes).
- **Type 1**: Email address, phone number (current state is sufficient).
- **Type 2**: Assigned sales region (historical accuracy required for sales attribution).

This hybrid approach balances storage efficiency with analytical requirements. When designing dimensions, evaluate each attribute individually to determine the appropriate change handling strategy.

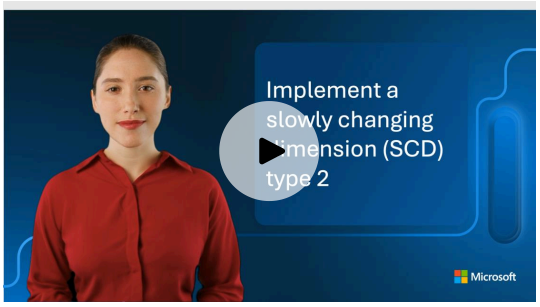
Understanding when to apply each SCD type enables you to implement dimensions that accurately support your organization's analytical requirements. The implementation details for loading and maintaining SCD tables in Azure Databricks are covered in later units.

7. Implement a slowly changing dimension (SCD) type 2

https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/7-implement-slowly-changing-dimension-type-2

Implement a slowly changing dimension (SCD) type 2

Completed



After selecting SCD Type 2 for a dimension, you need to implement the table structure and change capture logic in Azure Databricks. This unit focuses on creating SCD Type 2 tables and implementing the MERGE patterns that maintain version history as source data changes.

Create the SCD Type 2 table

The following SQL creates a customer dimension table with SCD Type 2 tracking columns:

```
CREATE TABLE sales.customer (
    customer_sk BIGINT GENERATED ALWAYS AS IDENTITY,
    customer_id STRING NOT NULL,
    customer_name STRING,
    email STRING,
    city STRING,
    region STRING,
    valid_from TIMESTAMP NOT NULL,
    valid_to TIMESTAMP NOT NULL,
    is_current BOOLEAN
)
USING DELTA
TBLPROPERTIES (
    delta.enableChangeDataFeed = true
);
```

The `GENERATED ALWAYS AS IDENTITY` clause creates an auto-incrementing surrogate key for each new row. Enabling the change data feed allows downstream processes to capture incremental changes efficiently.

Implement change capture with MERGE

The `MERGE` statement provides an efficient way to implement SCD Type 2 logic when processing updates from source systems. The statement handles inserts, updates, and the versioning logic in a single transaction.

The following pattern closes the current version and inserts the new version when changes occur:

```
MERGE INTO sales.customer AS target
USING (
```

```
    SELECT
        source.customer_id,
        source.customer_name,
        source.email,
        source.city,
        source.region,
        current_timestamp() AS valid_from,
        CAST('9999-12-31' AS TIMESTAMP) AS valid_to,
        true AS is_current
    FROM staging.customers AS source
) AS updates
ON target.customer_id = updates.customer_id AND target.is_current = true
WHEN MATCHED AND (
    target.customer_name <> updates.customer_name OR
    target.email <> updates.email OR
    target.city <> updates.city OR
    target.region <> updates.region
) THEN UPDATE SET
    target.valid_to = current_timestamp(),
    target.is_current = false
WHEN NOT MATCHED THEN INSERT (
    customer_id, customer_name, email, city, region,
    valid_from, valid_to, is_current
) VALUES (
    updates.customer_id, updates.customer_name, updates.email,
    updates.city, updates.region, updates.valid_from,
    updates.valid_to, updates.is_current
);

-- Insert new versions for updated records
INSERT INTO sales.customer
SELECT
    s.customer_id,
    s.customer_name,
    s.email,
    s.city,
    s.region,
    current_timestamp() AS valid_from,
    CAST('9999-12-31' AS TIMESTAMP) AS valid_to,
    true AS is_current
FROM staging.customers s
JOIN sales.customer h
    ON s.customer_id = h.customer_id
    AND h.valid_to = current_timestamp()
    AND h.is_current = false;
```

Tip

Consider using Lakeflow Spark Declarative Pipelines with the `AUTO CDC` API for automated SCD Type 2 processing. This approach handles out-of-order records and simplifies SCD Type 2 table maintenance. See the Azure Databricks documentation for change data capture pipelines.

Query historical data

Once you've implemented an SCD Type 2 table, you can query data as it existed at any point in time. The query pattern depends on your analytical needs.

Point-in-time queries

To find the state of data at a specific moment, filter on the validity period:

```
SELECT customer_id, customer_name, city
FROM sales.customer
WHERE valid_from <= '2023-06-15 12:00:00'
  AND valid_to > '2023-06-15 12:00:00';
```

This query returns exactly one row per customer—the version that was valid at the specified timestamp.

Track record history

To view the complete history of changes for a specific entity:

```
SELECT customer_name, city, valid_from, valid_to
FROM sales.customer
WHERE customer_id = 'C-555'
ORDER BY valid_from;
```

This query reveals all versions of customer C-555, showing how their attributes changed over time.

Use Delta Lake time travel

Delta Lake provides built-in time travel capabilities that complement explicit SCD Type 2 table designs. You can query previous table versions using the `TIMESTAMP AS OF` or `VERSION AS OF` syntax:

```
-- Query table state from 7 days ago
SELECT * FROM sales.customer
TIMESTAMP AS OF '2024-01-15';

-- Query a specific table version
SELECT * FROM sales.customer
VERSION AS OF 42;
```

> **Important**
>
> Delta Lake time travel has a default retention of 7 days. For longer historical analysis, use explicit SCD columns (ValidFrom, ValidTo) rather than relying solely on time travel. Configure

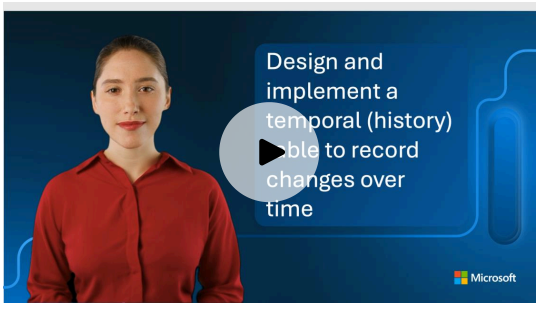
`delta.logRetentionDuration` and `delta.deletedFileRetentionDuration` if you need extended time travel access.

8. Design and implement a temporal (history) table to record changes over time

https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/8-design-temporal-history-table

Design and implement a temporal (history) table to record changes over time

Completed



Data pipelines often require tracking not just current values, but every change that occurs over time. Temporal tables provide a mechanism for recording row-level changes automatically, enabling point-in-time queries and complete audit trails. Unlike slowly changing dimensions that focus on business data history, temporal tables capture technical history for operational traceability and debugging.

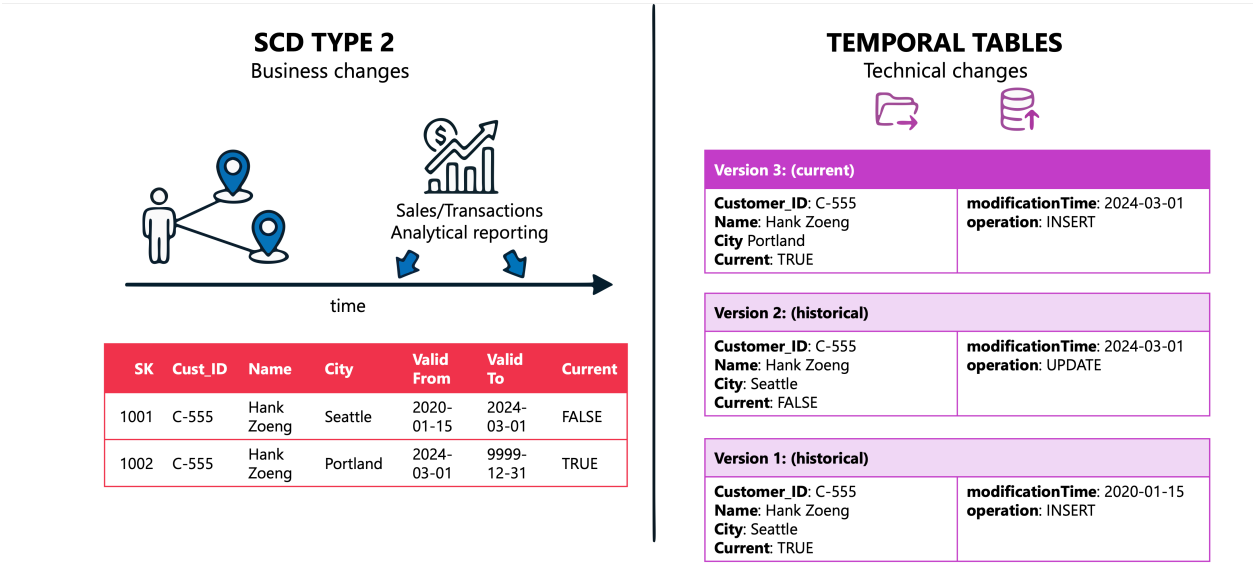
Understand temporal tables versus SCD Type 2

Temporal tables and SCD Type 2 dimensions serve different purposes, though both track historical changes. Understanding when to use each approach helps you design appropriate solutions.

SCD Type 2 tracks changes to business entities over time for analytical reporting. When a customer moves from Seattle to Portland, you want to attribute historical sales to the customer's location at the time of each transaction. SCD Type 2 maintains business-meaningful versions with explicit start and end dates.

Temporal (history) tables capture technical changes for operational needs. When troubleshooting why data loaded incorrectly, you need to see exactly what values existed at each point in time, which

files were processed, and when changes occurred. Temporal tables provide this forensic capability automatically.



Consider using temporal tables when you need to:

- Debug data ingestion issues by examining table state at specific timestamps
- Audit all changes for compliance or regulatory requirements
- Reproduce analyses or reports from a specific point in time
- Track operational metadata like load times and source file information

Enable change data feed for row-level tracking

Delta Lake's change data feed captures every row-level change automatically. When enabled, the runtime records change events with metadata indicating whether each row was inserted, deleted, or updated. Change data feed is disabled by default, so you must explicitly enable it on tables where you need this capability.

To enable change data feed on a new table:

```
CREATE TABLE bronze.sales_transactions (
    transaction_id STRING,
    customer_id STRING,
    amount DECIMAL(10,2),
    transaction_date TIMESTAMP,
    source_file STRING,
    load_timestamp TIMESTAMP
)
USING DELTA
TBLPROPERTIES (delta.enableChangeDataFeed = true);
```

For existing tables, enable change data feed with an `ALTER TABLE` statement:

```
ALTER TABLE bronze.sales_transactions
SET TBLPROPERTIES (delta.enableChangeDataFeed = true);
```

> **Important**
>
> Change data feed only records changes that occur after you enable it. Past changes to a table aren't captured. Enable this feature before loading production data.

Query table history

Once change data feed is enabled, you can query the change history to see exactly what modifications occurred. The change data feed adds three metadata columns to your data:

Column	Description
`_change_type`	The type of change: `insert`, `update_preimage`, `update_postimage`, or `delete`
`_commit_version`	The Delta log version containing the change
`_commit_timestamp`	The timestamp when the change was committed

Query changes within a specific time range:

```
SELECT
    transaction_id,
    amount,
    _change_type,
    _commit_version,
    _commit_timestamp
FROM table_changes('bronze.sales_transactions', '2026-01-01', '2026-01-31')
WHERE _change_type IN ('insert', 'update_postimage')
ORDER BY _commit_timestamp;
```

This query helps identify when specific records were modified, which is valuable for debugging data quality issues.

Use time travel for point-in-time queries

Delta Lake maintains a transaction log that enables querying data as it existed at any point in history. This capability supports debugging scenarios where you need to compare current data with a previous state.

Query data at a specific timestamp:

```
SELECT * FROM bronze.sales_transactions
```

```
TIMESTAMP AS OF '2026-01-15 14:30:00';
```

Query data at a specific version:

```
SELECT * FROM bronze.sales_transactions
VERSION AS OF 42;
```

To view the complete history of operations performed on a table:

```
DESCRIBE HISTORY bronze.sales_transactions;
```

The history output includes the operation type, timestamp, user, and detailed metrics for each change.

> **Note**
>
> Time travel availability depends on retention settings. By default, Delta Lake retains data files for 7 days and log files for 30 days. Configure `delta.deletedFileRetentionDuration` and `delta.logRetentionDuration` for longer retention periods.

Persist change history for long-term auditing

Because Delta Lake's built-in time travel has retention limits, you should persist change data to a separate history table for long-term audit requirements. This approach creates a permanent record of all changes.

Use Structured Streaming to continuously capture changes:

```
(spark.readStream
    .option("readChangeFeed", "true")
    .option("ignoreDeletes", "false")
    .table("bronze.sales_transactions")
    .writeStream
    .option("checkpointLocation", "/checkpoints/sales_history")
    .trigger(availableNow=True)
    .toTable("audit.sales_transactions_history")
)
```

The `readChangeFeed` option instructs Spark to read the change data feed, which returns row-level change events including inserts, updates, and deletes. The `ignoreDeletes` option controls whether delete operations are included in the stream. Setting it to `false` ensures that delete records are captured in the history table, providing a complete audit trail of all data changes.

The history table contains every change event with full metadata, enabling queries across the entire lifetime of your data—not just the retention window.

> **Tip**
>
> Schedule the history capture process to run after each data load. Using `trigger(availableNow=True)` processes all available changes as a batch, which works well with scheduled workflows.

Configure retention for compliance requirements

For temporal tables supporting compliance or regulatory needs, configure retention settings to match your requirements:

```
ALTER TABLE bronze.sales_transactions
SET TBLPROPERTIES (
    delta.logRetentionDuration = 'interval 90 days',
    delta.deletedFileRetentionDuration = 'interval 90 days'
);
```

Both properties must be set together. The `logRetentionDuration` controls how long history metadata is retained, while `deletedFileRetentionDuration` controls how long deleted data files are preserved before `VACUUM` removes them.

Longer retention periods increase storage costs but provide extended time travel access and change data feed availability for auditing.

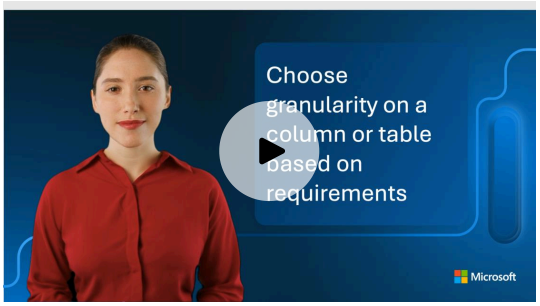
With temporal tables configured properly, you can trace every change to your data from ingestion through transformation, supporting both operational debugging and regulatory compliance requirements.

9. Choose granularity on a column or table based on requirements

https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/9-choose-granularity

Choose granularity on a column or table based on requirements

Completed



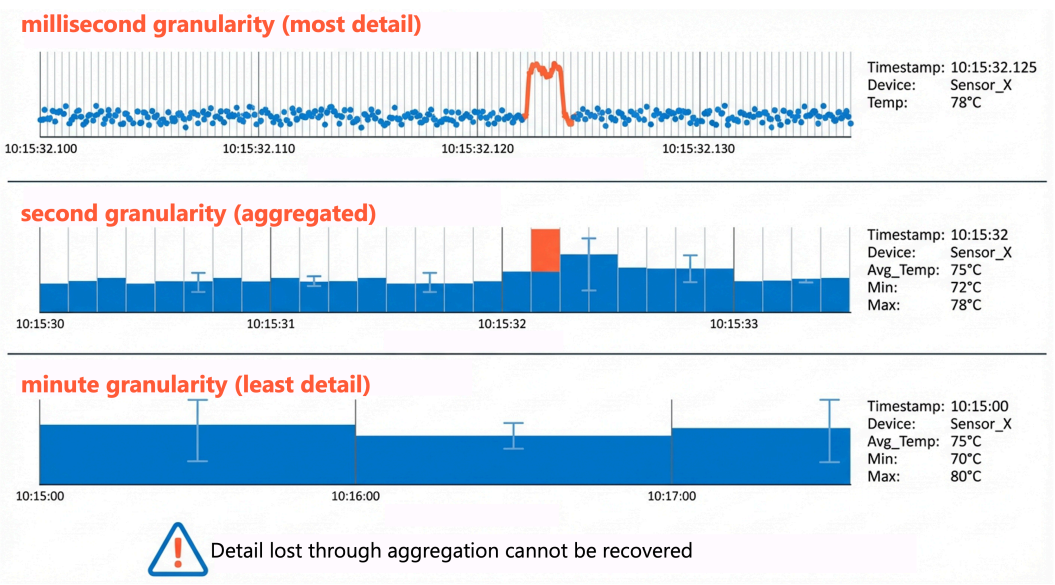
Granularity is one of the most critical design decisions you make when building a data model. It defines the **level of detail** captured in your tables and directly affects **storage costs**, **query performance**, and the **analytical questions** you can answer. As a data engineer, understanding granularity helps you design tables that meet business requirements while maintaining efficient operations.

This unit explains what granularity means in the context of dimensional modeling and how to evaluate different granularity options based on your organization's requirements.

Understand what granularity represents

Granularity, sometimes called **"grain,"** describes the level of detail that each row in a table represents. In a **fact table**, the grain determines what business event or measurement each row captures. In a **dimension table**, the grain determines how entities are represented and versioned.

Consider an IoT sensor data scenario. You could design your fact table at different levels of granularity:



- **Millisecond-level granularity**: Each row represents a single sensor reading captured at the millisecond level. This captures the most detailed information, including precise timestamps, sensor values, and device identifiers for each measurement.

- **Second aggregated granularity**: Each row represents the average, minimum, or maximum sensor value for a device per second. Individual readings are summarized into per-second statistics.
- **Minute aggregated granularity**: Each row represents aggregated sensor statistics for a device during an entire minute. This provides the least detail but requires the least storage.

The grain you choose affects everything downstream. Once you **lose detail through aggregation**, you **can't recover it**. If your minute-aggregated table shows that sensor X had an average temperature of 75°C during a specific minute, you can't determine which milliseconds had temperature spikes or identify the exact moment an anomaly occurred.

Evaluate factors that influence granularity decisions

Choosing the appropriate granularity requires balancing multiple competing concerns. The right choice depends on your specific business context and technical constraints.

Business requirements drive the grain

The analytical questions your organization needs to answer should be the **primary factor** in determining granularity. Ask yourself: what's the **most detailed level of analysis** required?

If operations engineers need to detect equipment anomalies at the millisecond level, per-minute aggregates won't suffice. If management only needs hourly utilization summaries by production line, storing every sensor reading wastes resources.

Consider these scenarios:

Business requirement	Minimum granularity needed
Detect equipment vibration anomalies in real-time	Millisecond event level
Analyze temperature trends for predictive maintenance	Second aggregate level
Monitor hourly energy consumption per device	Minute aggregate level
Report daily production metrics by facility	Hourly aggregate level

Query patterns shape performance

The way users query your data influences the optimal granularity. When queries typically filter and aggregate to higher levels, storing overly fine-grained data creates **unnecessary processing overhead**. Each query must scan and aggregate more rows than necessary.

At the same time, storing data at too coarse a granularity prevents users from drilling down to answer detailed questions. Finding the balance requires understanding both current and anticipated query patterns.

Storage and compute costs accumulate

Finer granularity means **more rows**, which increases **storage costs**. It also increases **compute costs** because queries must process more data. In Azure Databricks, these costs compound over time as historical data accumulates.

A manufacturing facility with thousands of sensors capturing readings at millisecond intervals might generate billions of rows daily at the finest granularity. Aggregating to per-second or per-minute statistics could reduce row counts by 99% or more, translating directly to lower storage and query costs.

Compliance and audit requirements matter

Regulatory requirements sometimes mandate specific granularity levels. Financial services organizations often must retain **transaction-level detail** for audit purposes, regardless of storage costs. Healthcare organizations might need to track individual patient interactions for compliance reasons.

When compliance drives your granularity decision, document the requirements clearly. This justification helps when stakeholders question why you're storing more detail than analytical needs alone would require.

Apply granularity to fact tables

Fact tables benefit most from careful granularity decisions because they typically contain the **largest volumes of data**. The grain of a fact table is determined by the combination of its **dimension keys**.

A sales fact table with dimension keys for date, product, customer, and store has a grain of "one row per product per customer per store per day." Every unique combination of these dimension values creates a distinct row. Adding a time dimension key at the hour level would change the grain to "one row per product per customer per store per hour," significantly increasing row counts.

Date_Key	Hour_Key	Product_Key	Customer_Key	Store_Key	Quantity_Sold	Sales_Amount
20260101	09	101	5001	10	1	20.00
20260101	14	101	5001	10	1	20.00
20260101	11	102	5001	10	1	25.00
20260101	16	101	5002	10	3	60.00

> **Important**
>
> Declare your fact table grain explicitly before adding any columns. The grain statement serves as a contract that guides all subsequent design decisions. For example: "This fact table contains one row per sales order line item."

When designing fact tables, consider whether you need:

- **Atomic grain**: The most detailed level captured in the source system. Choosing atomic grain provides **maximum analytical flexibility** because you can always aggregate up, but never

disaggregate down.

- **Aggregate grain**: Pre-summarized data at a higher level than the source. Aggregate fact tables **improve query performance** for common analytical patterns but limit the questions users can answer.

Many organizations maintain both atomic and aggregate fact tables, using aggregate tables as performance optimization while preserving atomic detail for ad-hoc analysis. In Azure Databricks, **materialized views** can automatically maintain these aggregate tables, refreshing them as the underlying atomic data changes.

Apply granularity to dimension tables

Dimension tables also have a grain, though it works differently than fact table grain. The dimension grain determines how you represent and version your business entities.

For a customer dimension, you might choose:

- **One row per customer**: The current state of each customer, updated in place when attributes change (SCD Type 1).
- **One row per customer per version**: A new row for each significant change, preserving historical states (SCD Type 2).
- **One row per customer per time period**: Periodic snapshots capturing customer attributes at regular intervals.

The dimension grain you choose must **align with your fact table grain**. If your fact table tracks individual transactions with surrogate keys to customer dimension, your customer dimension grain must support that relationship. A transaction referencing customer version 5 must be able to find that specific version in the dimension table.

Balance trade-offs when selecting granularity

No single granularity choice is universally correct. Each decision involves trade-offs that you must evaluate against your specific requirements.

Consideration	Fine granularity	Coarse granularity
Analytical flexibility	High - can answer detailed questions	Low - limited to aggregate analysis
Storage costs	Higher - more rows to store	Lower - fewer rows
Query performance	Slower for aggregate queries	Faster for common patterns
Historical analysis	Supports point-in-time analysis	May lose historical detail
Compliance support	Better for audit requirements	May not meet regulations

When you're uncertain, **starting with finer granularity** is often safer. You can always create aggregate tables later for performance optimization, but you **can't recover detail** that was never

captured. However, this guidance assumes you have the budget and infrastructure to support finer-grained storage initially.

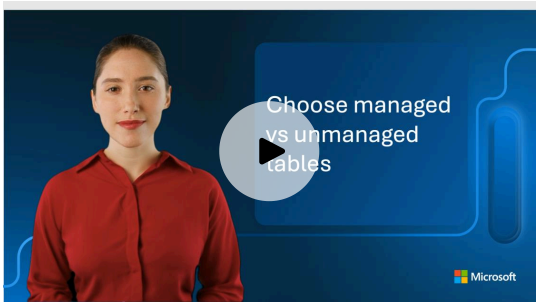
Understanding how to choose appropriate granularity enables you to design data models that support your organization's analytical needs while managing costs effectively. The implementation details for building and loading tables at your chosen granularity in Azure Databricks are covered in later units.

10. Choose managed vs unmanaged tables

https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/10-choose-managed-vs-unmanaged-tables

Choose managed vs unmanaged tables

Completed



When you create tables in Azure Databricks, you face a fundamental decision that affects data lifecycle management, governance, and operational flexibility. Understanding the differences between **managed tables** and **external tables** (sometimes called unmanaged tables) helps you choose the right approach for each scenario in your data platform.

In this unit, you learn how to distinguish between managed and external tables, and explore the factors that guide your choice.

Understand managed tables

Managed tables are fully controlled by Unity Catalog. This means Unity Catalog manages both the table metadata *and* the underlying data files in cloud storage. When you create a managed table without specifying a storage location, Azure Databricks automatically stores the data in a managed storage location associated with the containing schema, catalog, or metastore.

Because Unity Catalog has full control over managed tables, it can automatically optimize them for performance and cost. Several features are available exclusively for managed tables:

- **Predictive optimization** runs maintenance operations like compaction and vacuuming automatically
- **Automatic liquid clustering** intelligently selects and updates clustering keys based on query patterns
- **Metadata caching** provides faster query performance through in-memory caching
- **Automatic file cleanup** removes data files 8 days after you drop a table

Managed tables always use Delta Lake or Apache Iceberg format, ensuring ACID transactions and advanced data management capabilities.

Understand external tables

External tables (also called unmanaged tables) take a different approach. Unity Catalog manages only the table metadata, while you maintain control over the data files and their storage location. When you create an external table, you explicitly specify where the data resides using a `LOCATION` clause that points to an external location registered in Unity Catalog.

This separation means that when you drop an external table, Unity Catalog removes the metadata but leaves the data files intact in your storage location. You remain responsible for managing those files directly through your cloud provider.

External tables support a broader range of file formats beyond Delta Lake and Iceberg, including CSV, JSON, Avro, Parquet, ORC, and text files.

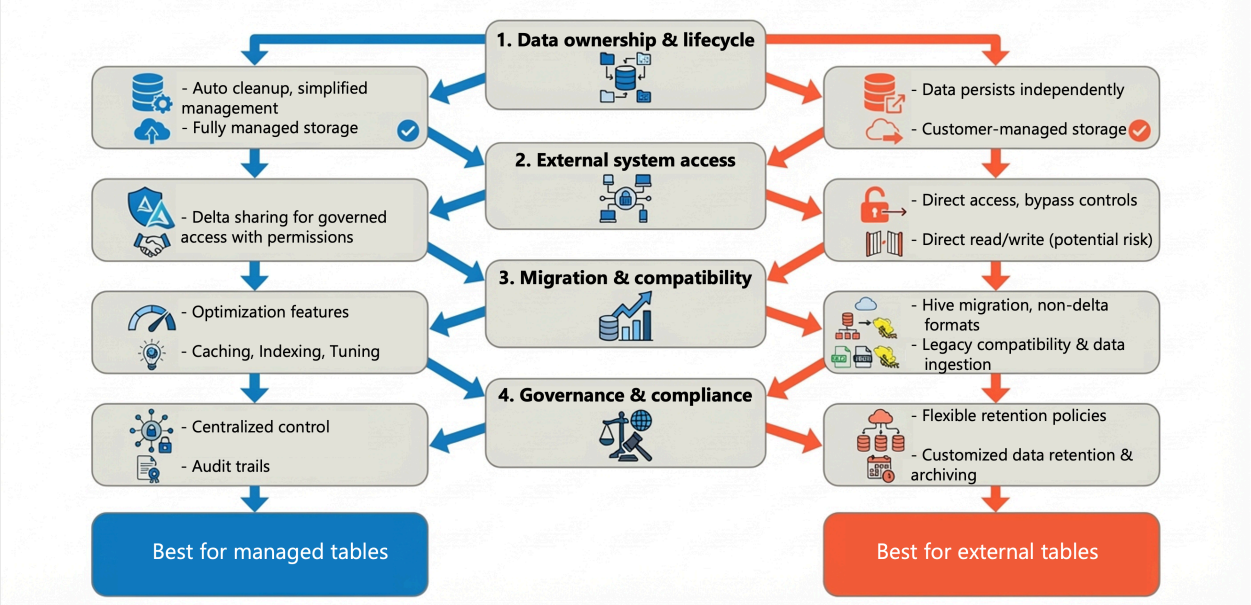
Compare key characteristics

The following table summarizes the key differences between managed and external tables:

Characteristic	Managed tables	External tables
Data storage location	Unity Catalog managed location	User-specified external location
Metadata management	Unity Catalog	Unity Catalog
Data file management	Unity Catalog	User managed
Supported formats	Delta Lake, Apache Iceberg	Delta, CSV, JSON, Avro, Parquet, ORC, text
Automatic optimization	Yes (predictive optimization)	No
Behavior on DROP TABLE	Files deleted after 8 days	Files remain in storage
UNDROP TABLE support	Yes (within 8 days)	No

Evaluate decision criteria

Choosing between managed and external tables depends on your specific requirements. Consider the following factors when making your decision.



Data ownership and lifecycle

If Azure Databricks should fully control the data lifecycle, including automatic cleanup when tables are dropped, use a managed table. This approach simplifies data management and reduces storage costs over time.

If the data must persist independently of the Azure Databricks workspace, or if you need to retain data after removing the table definition, use an external table. The data files remain available for other purposes even after the table is dropped.

External system access

When other tools or systems outside Azure Databricks need to read or write the data directly, external tables provide the necessary flexibility. However, consider that direct external access bypasses Unity Catalog access controls and auditing.

For sharing data across regions, cloud providers, or with external partners, **Delta Sharing** offers a governed approach that maintains security controls even with managed tables.

Migration and compatibility

External tables provide a practical path when migrating from Hive metastore to Unity Catalog. You can register existing data without moving files, allowing a quick upgrade. After migration, you can convert external tables to managed tables to take advantage of optimization features.

If you need to support non-Delta and non-Iceberg formats, external tables are your only option within Unity Catalog.

Governance and compliance

Consider your organization's data retention and compliance requirements. Managed tables offer stronger governance through centralized control, while external tables give you flexibility to manage data according to specific regulatory requirements or storage policies defined outside Azure Databricks.

> **Important**
>
> Avoid registering the same table as an external table in multiple metastores. This practice can lead to consistency issues because changes in one metastore don't automatically propagate to others.

Apply best practices

Databricks recommends managed tables for most use cases because they provide the best combination of governance, performance, and operational simplicity. Use external tables when your requirements specifically call for them.

When you do use external tables:

- Limit external access to reads when possible, with all writes happening through Azure Databricks
- Consider creating one external location per schema for clearer organization
- Plan to migrate external tables to managed tables when their original use case no longer applies

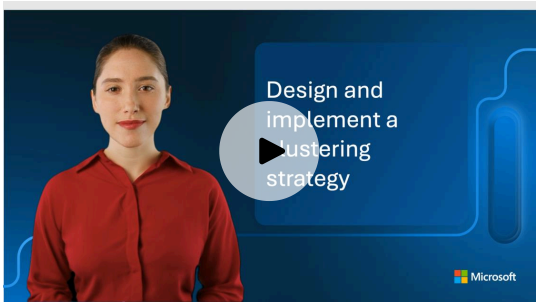
With this understanding of managed and external tables, you can make informed decisions that balance governance requirements, operational needs, and performance considerations for your data platform.

11. Design and implement a clustering strategy

https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/11-design-clustering-strategy

Design and implement a clustering strategy

Completed



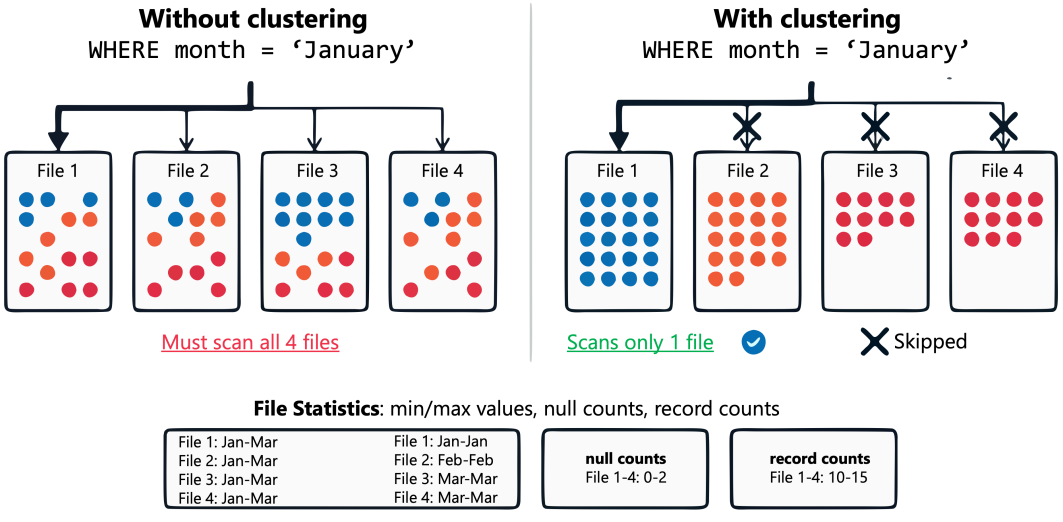
Optimizing how data is physically organized on disk directly affects **query performance** and **storage efficiency**. When you design a clustering strategy for your Delta Lake tables, you determine how the query engine locates and retrieves relevant data. The right approach can dramatically reduce the amount of data scanned during queries, lowering both response times and compute costs.

Azure Databricks provides several complementary techniques for data organization. Understanding when to apply **liquid clustering**, **Z-ordering**, or **deletion vectors** helps you make decisions that balance performance, maintenance effort, and compatibility with your existing tables.

Understand data skipping fundamentals

Before examining specific clustering techniques, it's important to understand the underlying mechanism they all leverage: **data skipping**. Delta Lake automatically collects statistics for each data file, including minimum and maximum values, null counts, and record counts for each column. When you execute a query with filter conditions, the query engine checks these statistics first and skips files that can't possibly contain matching records.

For example, if you query sales data for January 2026, and a particular file's statistics show it contains only records from March 2026, the engine skips that file entirely without reading its contents. This optimization happens automatically, but its effectiveness depends on how well your data is organized.



Data skipping becomes powerful when **related records are stored together** in the same files. If records for each month are scattered randomly across all files, the engine can't skip any files because

each file contains records from every month. Clustering techniques exist to solve exactly this problem.

Implement liquid clustering for new tables

Liquid clustering is the **recommended approach** for organizing data in new Delta Lake tables. Unlike traditional partitioning, liquid clustering allows you to **change your clustering strategy without rewriting existing data**. This flexibility is valuable when query patterns evolve or when you're uncertain which columns will be filtered most frequently.

To enable liquid clustering, use the `CLUSTER BY` clause when creating a table:

```
CREATE TABLE sales.transactions (
    transaction_id STRING,
    customer_id STRING,
    product_category STRING,
    region STRING,
    transaction_date DATE,
    amount DECIMAL(12,2)
)
CLUSTER BY (region, transaction_date);
```

You can specify **up to four clustering keys**. Choose columns that appear frequently in your query filters. The order of clustering keys matters less than selecting the right columns, as liquid clustering optimizes for all specified keys simultaneously.

After creating a clustered table, run the `OPTIMIZE` command to apply clustering to your data:

```
OPTIMIZE sales.transactions;
```

Liquid clustering is **incremental**. Each `OPTIMIZE` run clusters only the data that needs reorganization, making regular maintenance efficient. Databricks recommends scheduling `OPTIMIZE` jobs every one or two hours for tables with frequent inserts or updates.

> **Tip**
>
> For Unity Catalog managed tables, enable **predictive optimization** to automate `OPTIMIZE` operations. Azure Databricks analyzes your workload patterns and runs maintenance operations automatically.

Change clustering keys when needed

One significant advantage of liquid clustering over partitioning is the ability to modify your clustering strategy. If your query patterns change, update the clustering keys with an `ALTER TABLE` statement:

```
ALTER TABLE sales.transactions
CLUSTER BY (customer_id, transaction_date);
```

Subsequent `OPTIMIZE` operations use the new clustering approach. Previously written data isn't automatically rewritten. To apply new clustering to all existing data, run:

```
OPTIMIZE sales.transactions FULL;
```

Enable automatic clustering key selection

Azure Databricks can intelligently select clustering keys based on your actual query patterns. Automatic liquid clustering analyzes workload history and chooses columns that provide the greatest data skipping benefit:

```
CREATE TABLE sales.orders CLUSTER BY AUTO;
```

You can enable automatic clustering on existing tables:

```
ALTER TABLE sales.transactions CLUSTER BY AUTO;
```

Automatic liquid clustering requires predictive optimization to be enabled. Azure Databricks periodically evaluates whether different clustering keys would improve performance and adjusts the strategy accordingly.

Migrate from Z-ordering to liquid clustering

Z-ordering is a legacy technique that organizes data to colocate related information in the same files. While liquid clustering is recommended for new tables, you'll encounter existing tables that use Z-ordering and need to understand both approaches.

Z-ordering uses the `OPTIMIZE` command with a `ZORDER BY` clause:

```
OPTIMIZE sales.legacy_transactions
ZORDER BY (region, transaction_date);
```

Z-ordering differs from liquid clustering in several important ways:

Aspect	Liquid Clustering	Z-Ordering
Key flexibility	Change keys without data rewrite	Changing keys requires full rewrite
Write performance	Clustering on write for eligible operations	Applied only during OPTIMIZE
Concurrent writes	Row-level concurrency supported	Transaction-level concurrency
Compatibility	Requires Databricks Runtime 13.3 LTS or above	Supported in all runtimes

When you inherit a Z-ordered table, you have two options. Continue using Z-ordering for consistency, or migrate to liquid clustering for improved flexibility. To migrate, first enable liquid clustering on the existing unpartitioned table:

```
ALTER TABLE sales.legacy_transactions
CLUSTER BY (region, transaction_date);
```

Then run a full optimization to reorganize all existing data:

```
OPTIMIZE sales.legacy_transactions FULL;
```

> **Important**
>
> Liquid clustering is not compatible with partitioning. For partitioned tables, evaluate whether the benefits of liquid clustering justify migrating to an unpartitioned table structure.

Manage deletion vectors for efficient updates

Deletion vectors address a different aspect of data organization: how Delta Lake handles modified or deleted records. By default, when you update or delete a single row in a Parquet file, Delta Lake must **rewrite the entire file**. For tables with frequent modifications, this creates significant overhead.

With deletion vectors enabled, Delta Lake **marks modified rows as deleted** in a separate metadata file rather than rewriting the Parquet file immediately. Subsequent reads apply these deletions on the fly, combining the original file with the deletion vector to return accurate results.

Deletion vectors are enabled by default for new tables created with Databricks Runtime 14.1 or above. You can enable them explicitly on existing tables:

```
ALTER TABLE sales.transactions
SET TBLPROPERTIES ('delta.enableDeletionVectors' = true);
```

Deletion vectors are applied physically when the data files are rewritten during:

- An `OPTIMIZE` command
- Auto-compaction triggers
- A `REORG TABLE ... APPLY (PURGE)` operation

For compliance scenarios where you need to ensure deleted data is **physically removed**, run a `REORG` operation followed by `VACUUM`:

```
REORG TABLE sales.transactions APPLY (PURGE);
VACUUM sales.transactions;
```

Deletion vectors also enable **row-level concurrency**, allowing multiple concurrent transactions to modify different rows in the same table without conflicts. This capability is particularly valuable for tables with high update volumes.

12. Exercise - Design and Implement Data Modeling with Azure Databricks

https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/12-exercise-design-implement-data-modeling-unity-catalog

Exercise - Design and Implement Data Modeling with Azure Databricks

Completed

Now it's your chance to design and implement data modeling with Azure Databricks. In this lab, you design and implement a Delta Lake data model in Unity Catalog for a retail banking scenario, building a customer dimension with SCD Type 2 history tracking and a transaction fact table with liquid clustering. You apply Change Data Feed to build a queryable FCA compliance audit trail and use Delta Lake time travel to inspect and restore previous table versions.

> **Note**
>
> To complete this lab, you need an Azure subscription in which you have administrative access.

Launch the exercise and follow the instructions.

Launch Exercise

13. Knowledge check

https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/13-knowledge-check

Knowledge check

Completed

14. Summary

https://learn.microsoft.com/en-us/training/modules/design-implement-data-modeling-unity-catalog/14-summary

Summary

Completed

You've learned how to design and implement data modeling strategies that form the foundation of a well-architected Azure Databricks lakehouse. Starting with ingestion logic design, you explored how to match extraction types to source system capabilities and latency requirements. You evaluated ingestion tools—from Lakeflow Connect for managed connectors to Auto Loader for file-based streaming—and selected the right tool for each scenario.

Table format selection between Delta Lake and Apache Iceberg depends on your requirements: Delta Lake for deep Azure Databricks integration and Apache Iceberg when cross-platform compatibility is essential. You designed partitioning schemes that enable partition pruning for tables exceeding 1 TB. Slowly changing dimension types give you the tools to maintain historical accuracy where business requirements demand it.

Granularity decisions establish analytical flexibility. Starting with atomic grain preserves maximum detail, while aggregate tables optimize common query patterns. Liquid clustering organizes data for efficient data skipping, and managed tables simplify operations through predictive optimization.

Apply these concepts by evaluating your current data pipelines against the decision frameworks presented. Start with one table that would benefit from improved clustering, measure performance before and after, and expand your optimizations based on results.

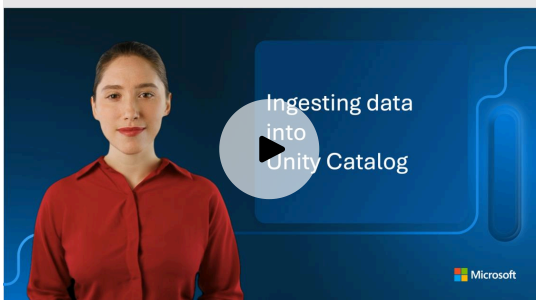
Ingest data into Unity Catalog

1. Introduction

https://learn.microsoft.com/en-us/training/modules/ingest-data-into-unity-catalog/1-introduction

Introduction

Completed



Data engineers face a fundamental challenge: getting data from diverse sources into a unified analytics platform efficiently and reliably. Whether your organization works with batch files in cloud storage, streaming events from Kafka, or transactional data from operational databases, you need ingestion methods that handle each scenario while maintaining data quality and governance. **Unity Catalog** in Azure Databricks provides the foundation for this unified approach, offering centralized governance for all your ingested data.

Azure Databricks offers multiple ingestion techniques, each optimized for specific data patterns and requirements. **Lakeflow Connect** provides managed connectors for common enterprise sources like SQL Server and Salesforce, eliminating the need for custom extraction code. **Notebooks** give you full control over ingestion logic using Python, SQL, or Scala. **SQL commands** like `COPY INTO` and `CREATE TABLE AS SELECT` offer declarative approaches for file-based ingestion. For real-time workloads, **Spark Structured Streaming** and **Auto Loader** process data continuously as it arrives, with exactly-once processing guarantees.

Understanding when to apply each technique enables you to build robust data pipelines that scale with your organization. You'll learn how to configure managed connectors for enterprise data sources, write notebook code for custom ingestion scenarios, use SQL commands for file-based batch loads, process change data capture feeds incrementally, stream data from message buses like Kafka and Event Hubs, configure Auto Loader for automatic file detection, and orchestrate ingestion workflows with Lakeflow Spark Declarative Pipelines.

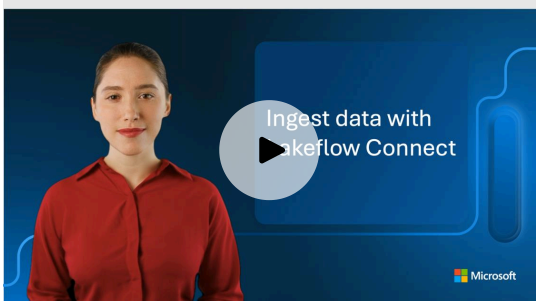
By the end of this module, you'll be equipped to choose and implement the right ingestion approach for any data source, ensuring your data lands in Unity Catalog tables with proper governance and optimal performance.

2. Ingest data with Lakeflow Connect

https://learn.microsoft.com/en-us/training/modules/ingest-data-into-unity-catalog/2-ingest-data-lakeflow-connect

Ingest data with Lakeflow Connect

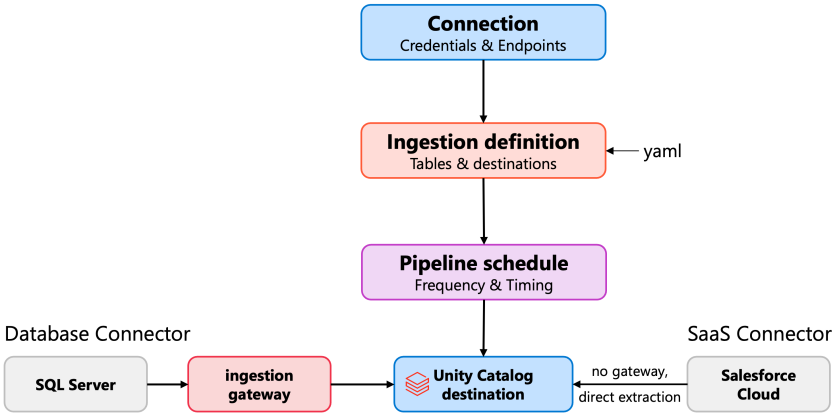
Completed



Data engineers face a common challenge: getting data from diverse sources into analytics platforms efficiently and reliably. Whether your data lives in SQL Server, Salesforce, SharePoint, or cloud storage, you need consistent, governed pipelines that minimize manual effort. **Lakeflow Connect** addresses this challenge by providing managed ingestion connectors that bring data directly into Unity Catalog tables.

Understand ingestion pipelines

Lakeflow Connect is a collection of managed connectors in Azure Databricks that simplify data ingestion from external sources. Rather than writing custom extraction code, you configure pipelines through either a graphical interface or declarative definitions.



Each pipeline consists of three core components:

- **Connection**: Stores credentials and endpoint information for the source system. You create connections once and reuse them across multiple pipelines.
- **Ingestion definition**: Specifies which tables or objects to extract, along with their destination catalogs and schemas in Unity Catalog.
- **Pipeline schedule**: Controls when and how frequently data flows from source to destination.

When you create an ingestion pipeline for a database connector like SQL Server, Lakeflow Connect also creates an **ingestion gateway**. This gateway continuously extracts change data from the source database and stages it for processing. For SaaS connectors like Salesforce, the connector handles extraction directly without a separate gateway.

Configure ingestion pipelines

You can create ingestion pipelines through the Databricks UI, Asset Bundles, notebooks, or the CLI. The UI provides the simplest approach for initial setup.

Consider a scenario where you need to ingest customer order data from SQL Server. In the Azure Databricks workspace, you navigate to **Data Ingestion** and select **SQL Server**. The wizard guides you through creating an ingestion gateway, selecting the connection, and choosing the tables to ingest. You specify a destination catalog and schema where the ingested data lands as streaming tables.

```
resources:
  pipelines:
    pipeline_sqlserver:
      name: customer-orders-pipeline
      catalog: sales
      schema: bronze
      ingestion_definition:
        ingestion_gateway_id: ${resources.pipelines.gateway.id}
        objects:
          - table:
              source_schema: dbo
              source_table: orders
```

```
            destination_catalog: sales
            destination_schema: bronze
```

With this configuration, the pipeline extracts the `orders` table from the source database and loads it into `sales.bronze.orders` in Unity Catalog.

Control data extraction behavior

Lakeflow Connect offers several options for controlling how data flows from source to destination. The most important choice is between **snapshot** and **incremental** extraction patterns.

For database connectors, the gateway uses **change data capture (CDC)** to track modifications at the source. When CDC is enabled on the source database, the gateway captures inserts, updates, and deletes incrementally. This means subsequent pipeline runs process only changed records rather than reloading entire tables.

The **SCD type** setting determines how the destination table handles changes:

- **SCD type 1**: Overwrites existing records when updates occur. The destination table always reflects the current state of the source data.
- **SCD type 2**: Preserves historical versions of records. When a row changes, the pipeline marks the previous version as inactive and adds the new version. This approach tracks how data evolves over time.

```
table_configuration:
  scd_type: SCD_TYPE_2
  sequence_by: last_modified
```

When you enable SCD type 2, the destination table includes `__START_AT` and `__END_AT` columns that record each version's active period.

Optimize column selection and refresh strategies

Not every column in your source tables may be relevant for analytics. Lakeflow Connect lets you specify exactly which columns to include or exclude from ingestion.

```
table_configuration:
  include_columns:
    - customer_id
    - order_date
    - total_amount
```

With explicit column selection, you reduce data transfer volumes and storage costs. If you use `include_columns`, newly added source columns won't automatically appear in the destination. With `exclude_columns`, new columns are included by default.

For situations where incremental processing can't recover from schema changes or data inconsistencies, you can trigger a **full refresh**. This operation clears the destination table and reloads all records from scratch. Lakeflow Connect minimizes downtime during full refresh by maintaining the existing data until the new snapshot completes.

You can also configure a **full refresh window** to schedule when these operations occur, avoiding disruption during peak business hours.

Scale with multiple destinations

A single ingestion pipeline can write to multiple destination catalogs and schemas. This capability proves useful when different teams need isolated copies of the same source data.

```
objects:
  - table:
      source_table: customers
      destination_catalog: sales
      destination_schema: analytics
  - table:
      source_table: customers
      destination_catalog: marketing
      destination_schema: campaigns
```

You can even ingest the same source table to multiple destinations within the same schema by specifying a custom `destination_table` name.

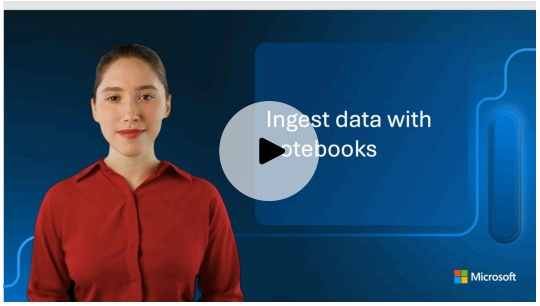
Once your pipelines are running, you can monitor ingestion progress, set up failure notifications, and adjust schedules through the pipeline details page. Lakeflow Connect automatically creates jobs for each schedule you define, allowing you to add additional processing tasks after ingestion completes.

3. Ingest data with notebooks

https://learn.microsoft.com/en-us/training/modules/ingest-data-into-unity-catalog/3-ingest-data-notebooks

Ingest data with notebooks

Completed



Notebooks in Azure Databricks provide a flexible, code-driven approach for ingesting data from various sources. When graphical tools don't meet your requirements, or when you need custom logic for data transformation, notebooks give you complete control over the ingestion process. You can use Python (PySpark), SQL, Scala, or R to connect to data sources, apply transformations, and load data into Unity Catalog tables.

In this unit, you learn how to ingest data using notebooks for both batch and streaming scenarios.

Ingest batch data with DataFrames

DataFrames are the foundation for batch data ingestion in Databricks. A DataFrame is a distributed collection of data organized into named columns, similar to a table in a relational database. You use the `spark.read` API to load data from various sources into a DataFrame.

The following example reads a CSV file into a DataFrame:

```
df = (spark.read
    .format("csv")
    .option("header", "true")
    .option("inferSchema", "true")
    .load("/Volumes/catalog/schema/volume/data.csv"))
```

The `format()` method specifies the file type, while `option()` methods configure how the data is parsed. The `header` option tells Spark to use the first row as column names, and `inferSchema` automatically detects data types.

Read from different file formats

Azure Databricks supports multiple file formats. You can adjust the format and options based on your data source:

```
# Read JSON files
df_json = spark.read.format("json").load("/path/to/data.json")

# Read Parquet files (columnar format, efficient for analytics)
df_parquet = spark.read.format("parquet").load("/path/to/data.parquet")

# Read XML files (requires rowTag option)
df_xml = (spark.read
```

```
    .format("xml")
    .option("rowTag", "record")
    .load("/path/to/data.xml"))
```

> **Note**
>
> The `rowTag` option specifies which XML element represents a row. For example, given this XML structure:
>
> ```
> <data>
>   <record><id>1</id><name>Alice</name></record>
>   <record><id>2</id><name>Bob</name></record>
> </data>
> ```
>
> Setting `rowTag` to `"record"` tells Spark to treat each `<record>` element as a single row in the DataFrame.

Parquet files store data in a columnar format, which improves query performance for analytical workloads. JSON files support nested structures and are common for API responses. XML files require the `rowTag` option to specify which element maps to each row.

Connect to external databases

For data stored in relational databases, use JDBC connections to read directly from the source:

```
df_jdbc = (spark.read
    .format("jdbc")
    .option("url", "jdbc:sqlserver://server.database.windows.net:1433;database=mydb
    .option("dbtable", "schema.tablename")
    .option("user", dbutils.secrets.get(scope="jdbc", key="username"))
    .option("password", dbutils.secrets.get(scope="jdbc", key="password"))
    .load())
```

Store credentials in Databricks secrets rather than hardcoding them in notebooks. You can also push queries down to the database by using a subquery in the `dbtable` option:

```
pushdown_query = "(SELECT id, name, amount FROM orders WHERE order_date > '2024-01-
df = spark.read.format("jdbc").option("dbtable", pushdown_query).load()
```

Write data to Unity Catalog tables

After loading and transforming your data, write it to a Unity Catalog table. Azure Databricks uses Delta Lake format by default, which provides ACID transactions and versioning:

```
df.write.mode("overwrite").saveAsTable("catalog.schema.table_name")
```

The `mode()` option controls how existing data is handled:

- **overwrite**: Replaces the table contents entirely
- **append**: Adds rows to the existing table
- **ignore**: Does nothing if the table already exists
- **error**: Throws an exception if the table exists (default)

Ingest streaming data with Structured Streaming

Structured Streaming enables you to process data continuously as it arrives. Instead of `spark.read`, you use `spark.readStream` to create a streaming DataFrame that processes new data incrementally.

The following example reads streaming data from Kafka:

```
df_stream = (spark.readStream
    .format("kafka")
    .option("kafka.bootstrap.servers", "broker:9092")
    .option("subscribe", "topic-name")
    .option("startingOffsets", "latest")
    .load())
```

The `startingOffsets` option determines where to begin reading when starting a new query. Use `latest` for new data only or `earliest` to read from the beginning of the topic. Once you configure a checkpoint location in the write stream, subsequent runs automatically resume from the last processed offset, ignoring `startingOffsets`.

Process file-based streams

You can also stream from file sources as new files arrive in a directory:

```
df_file_stream = (spark.readStream
    .format("csv")
    .option("header", "true")
    .schema(defined_schema)
    .load("/Volumes/catalog/schema/volume/incoming/"))
```

For streaming file sources, you should define the schema explicitly rather than using `inferSchema`. Schema inference isn't supported for streaming sources because it requires reading the entire dataset.

Write streaming data to tables

Use `writeStream` to persist streaming data to Unity Catalog tables:

```
(df_stream
    .writeStream
    .format("delta")
```

```
    .outputMode("append")
    .option("checkpointLocation", "/checkpoints/my-stream")
    .toTable("catalog.schema.streaming_table"))
```

The output mode determines which records are written:

- **append**: Only new rows are written (default for most operations)
- **update**: Changed rows are written
- **complete**: All rows are rewritten (used with aggregations)

The checkpoint location stores progress information, allowing the stream to resume from where it stopped if interrupted.

Handle semi-structured data

When working with semi-structured data like JSON, the VARIANT data type provides flexibility for storing and querying nested structures:

```
from pyspark.sql.functions import parse_json, variant_get, col

df = spark.read.text("/path/to/data.json")
df_variant = df.select(parse_json(col("value")).alias("data"))
```

You can then query specific fields using the `variant_get` function or SQL syntax:

```
df_variant.select(variant_get(col("data"), "$.customer.name", "string")).display()
```

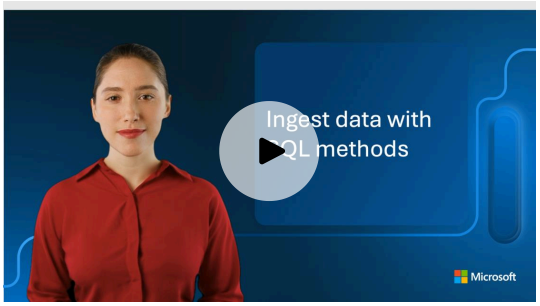
This approach works well when the JSON schema varies between records or evolves over time.

4. Ingest data with SQL methods

https://learn.microsoft.com/en-us/training/modules/ingest-data-into-unity-catalog/4-ingest-data-sql-methods

Ingest data with SQL methods

Completed



SQL commands provide a declarative approach to data ingestion in Azure Databricks. If you're already comfortable with SQL syntax, these methods let you create and populate tables without writing procedural code. The three primary SQL ingestion techniques— `CREATE TABLE AS SELECT (CTAS)`, `CREATE OR REPLACE TABLE`, and `COPY INTO` —each address different ingestion scenarios while maintaining full compatibility with Unity Catalog.

Create tables from queries with CTAS

The `CREATE TABLE AS SELECT` statement combines table creation and data population into a single operation. You define a new table based on the results of a `SELECT` query, making it ideal for transforming data during ingestion.

Consider a scenario where you need to ingest customer data from an external source and apply transformations. With CTAS, you write a query that reads from the source, applies your transformations, and saves the results to a new managed table:

```
CREATE TABLE catalog.schema.customers AS
SELECT
    customer_id,
    UPPER(customer_name) AS customer_name,
    email,
    created_date
FROM external_staging.raw_customers
WHERE customer_status = 'active';
```

The table schema is automatically derived from the query results. Azure Databricks creates the table using Delta format by default, which provides ACID transactions, time travel, and optimized performance.

CTAS works well for initial data loads and one-time migrations. When you need to read data from files, combine CTAS with the `read_files` table-valued function:

```
CREATE TABLE catalog.schema.sales_data AS
SELECT * FROM read_files(
    '/Volumes/catalog/schema/volume/sales/*.parquet',
    format => 'parquet'
);
```

> **Note**

> CTAS creates a new table each time it runs. If the table already exists, the command fails unless you use the `IF NOT EXISTS` clause—but that clause skips execution entirely rather than updating the table.

Refresh tables with CREATE OR REPLACE TABLE

When you need to fully refresh a table's contents, `CREATE OR REPLACE TABLE` provides a clean solution. This command either creates a new table or completely replaces an existing one, preserving table history, granted privileges, and any row filters or column masks you've configured.

This approach proves valuable for periodic data refreshes where you want to replace all existing data:

```
CREATE OR REPLACE TABLE catalog.schema.daily_metrics AS
SELECT
    report_date,
    SUM(revenue) AS total_revenue,
    COUNT(DISTINCT customer_id) AS unique_customers
FROM catalog.schema.transactions
WHERE report_date >= CURRENT_DATE - INTERVAL 30 DAYS
GROUP BY report_date;
```

Unlike dropping and recreating a table, `CREATE OR REPLACE` maintains the table's metadata and permissions. Downstream users and applications continue to access the table without reconfiguration.

You can also use this command to load data directly from files. The schema is automatically inferred from the query results:

```
CREATE OR REPLACE TABLE catalog.schema.products AS
SELECT * FROM read_files(
    '/Volumes/catalog/schema/volume/products.csv',
    format => 'csv',
    header => true
);
```

> **Important**
>
> `CREATE OR REPLACE` performs a full table replacement. For incremental updates where you only want to add new records, use `COPY INTO` instead.

Load files incrementally with COPY INTO

`COPY INTO` addresses a common ingestion challenge: loading files from cloud storage in a reliable, repeatable manner. Unlike CTAS, which runs once and creates a table, `COPY INTO` is designed for ongoing ingestion workflows where new files arrive regularly.

The command reads files from a specified location and appends them to an existing Delta table. Its key feature is idempotency—files that have already been loaded are automatically skipped, even across multiple executions:

```
COPY INTO catalog.schema.events
FROM '/Volumes/catalog/schema/volume/events/'
FILEFORMAT = JSON
FORMAT_OPTIONS ('multiline' = 'true');
```

Before running `COPY INTO`, the target table must already exist. Create it with the appropriate schema:

```
CREATE TABLE IF NOT EXISTS catalog.schema.events (
    event_id STRING,
    event_type STRING,
    event_timestamp TIMESTAMP,
    payload STRING
);
```

Configure file selection

When your source directory contains files with different naming patterns or you need to load specific files, use the `PATTERN` or `FILES` options:

```
-- Load only files matching a pattern
COPY INTO catalog.schema.orders
FROM '/Volumes/catalog/schema/volume/orders/'
FILEFORMAT = PARQUET
PATTERN = 'orders_2024*.parquet';

-- Load specific files by name
COPY INTO catalog.schema.orders
FROM '/Volumes/catalog/schema/volume/orders/'
FILEFORMAT = PARQUET
FILES = ('orders_001.parquet', 'orders_002.parquet');
```

Handle schema and data quality

`COPY INTO` provides options for handling schema changes and validating data before loading:

```
COPY INTO catalog.schema.sensor_data
FROM '/Volumes/catalog/schema/volume/sensors/'
FILEFORMAT = CSV
FORMAT_OPTIONS (
    'header' = 'true',
    'inferSchema' = 'true'
)
COPY_OPTIONS ('mergeSchema' = 'true');
```

The `mergeSchema` option allows the table schema to evolve as new columns appear in source files. For validating data without loading it, add the VALIDATE clause:

```
COPY INTO catalog.schema.sensor_data
FROM '/Volumes/catalog/schema/volume/sensors/'
FILEFORMAT = CSV
VALIDATE ALL;
```

This validation checks whether data can be parsed, matches the table schema, and satisfies nullability and check constraints.

Choose the right method

Each SQL ingestion method serves distinct purposes in your data engineering workflows:

Method	Best for	Behavior
CTAS	Initial data loads, one-time migrations, creating tables from queries	Creates a new table; fails if table exists
CREATE OR REPLACE	Periodic full refreshes, replacing staging tables	Replaces entire table; preserves permissions
COPY INTO	Ongoing file ingestion, incremental loads	Appends to existing table; skips loaded files

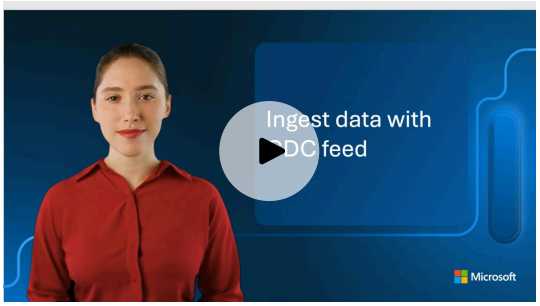
For file-based ingestion that requires automatic schema inference, file notifications, or exactly-once guarantees, consider using Auto Loader as a complementary approach. When your ingestion needs are straightforward and you prefer declarative SQL over procedural code, these three methods provide a complete toolkit for managing data flow into Unity Catalog.

5. Ingest data with CDC feed

https://learn.microsoft.com/en-us/training/modules/ingest-data-into-unity-catalog/5-ingest-data-change-data-capture-feed

Ingest data with CDC feed

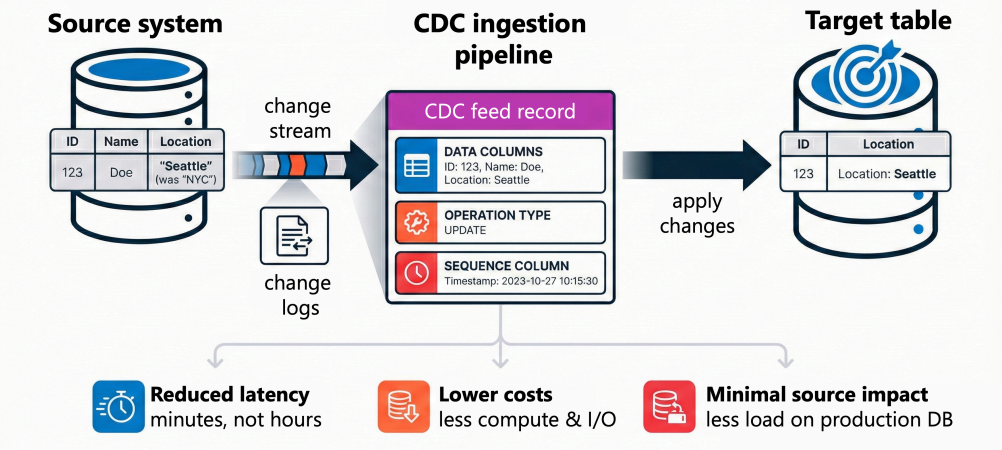
Completed



When source systems generate millions of transactions daily, reloading entire tables becomes impractical. You need a way to capture only the changes—inserts, updates, and deletes—and apply them efficiently to your destination tables. This pattern, known as change data capture (CDC), forms the foundation of modern incremental data pipelines.

Understand CDC ingestion patterns

Change data capture ingestion involves reading a stream of change records from a source system and applying those changes to a target table. Unlike full table reloads, CDC processes only what has changed since the last sync, reducing both processing time and resource consumption.



A typical CDC feed contains records with the following structure:

- **Data columns**: The actual values for each field in the source table
- **Operation type**: Indicates whether the record is an INSERT, UPDATE, or DELETE
- **Sequence column**: A timestamp or sequence number that determines the order of changes

Consider an employee table that tracks name and location. When you update an employee's city, the CDC feed doesn't send the entire table—it sends a single record with the new city value, the operation type UPDATE, and a sequence number. Your ingestion pipeline reads this record and applies the change to the corresponding row in the destination table.

This approach offers several advantages for data engineers:

- **Reduced latency**: Changes flow to the destination within minutes rather than hours
- **Lower costs**: Processing fewer records means less compute time and storage I/O

- **Minimal source impact**: Reading change logs puts less load on production databases than full table scans

Process CDC with the AUTO CDC API

Azure Databricks provides the AUTO CDC API in Lakeflow Spark Declarative Pipelines to simplify change data processing. This API handles the complexity of out-of-order records, deduplication, and change application automatically.

To implement CDC processing, you first create a streaming table as the target. Then you define a flow that reads change records from your source and applies them using the appropriate slowly changing dimension (SCD) type.

SCD Type 1 keeps only the current version of each record:

```
from pyspark import pipelines as dp
from pyspark.sql.functions import col, expr

@dp.view
def employees_cdf():
    return spark.readStream.table("bronze.employee_changes")

dp.create_streaming_table("employees")

dp.create_auto_cdc_flow(
    target="employees",
    source="employees_cdf",
    keys=["employee_id"],
    sequence_by=col("sequence_num"),
    apply_as_deletes=expr("operation = 'DELETE'"),
    except_column_list=["operation", "sequence_num"],
    stored_as_scd_type=1
)
```

With SCD Type 1, when an employee moves from Seattle to Portland, the destination table updates the city column directly. The Seattle record disappears—only Portland remains.

SCD Type 2 preserves the complete history of changes:

```
dp.create_streaming_table("employees_history")

dp.create_auto_cdc_flow(
    target="employees_history",
    source="employees_cdf",
    keys=["employee_id"],
    sequence_by=col("sequence_num"),
    apply_as_deletes=expr("operation = 'DELETE'"),
    except_column_list=["operation", "sequence_num"],
    stored_as_scd_type=2
)
```

SCD Type 2 adds `__START_AT` and `__END_AT` columns to track when each version was active. The Seattle record remains with an end date, while a new Portland record appears with no end date. Analysts can query the table to see exactly when and how employee data changed over time.

Configure CDC flows in SQL

The same CDC patterns work in SQL using declarative syntax. SQL flows integrate naturally with medallion architecture pipelines where you define bronze, silver, and gold layers.

```
CREATE OR REFRESH STREAMING TABLE employees;

CREATE FLOW employees_cdc_flow
AS AUTO CDC INTO employees
FROM stream(bronze.employee_changes)
KEYS (employee_id)
APPLY AS DELETE WHEN operation = 'DELETE'
SEQUENCE BY sequence_num
COLUMNS * EXCEPT (operation, sequence_num)
STORED AS SCD TYPE 1;
```

The SQL syntax mirrors the Python API. You specify the target table, source stream, key columns, delete condition, sequencing column, and SCD type. Columns you don't need in the destination—like the operation and sequence columns—get excluded with the `EXCEPT` clause.

For historical tracking with SCD Type 2, you can also limit which columns trigger new history records:

```
CREATE FLOW employees_history_flow
AS AUTO CDC INTO employees_history
FROM stream(bronze.employee_changes)
KEYS (employee_id)
SEQUENCE BY sequence_num
COLUMNS * EXCEPT (operation, sequence_num)
STORED AS SCD TYPE 2
TRACK HISTORY ON * EXCEPT (last_login);
```

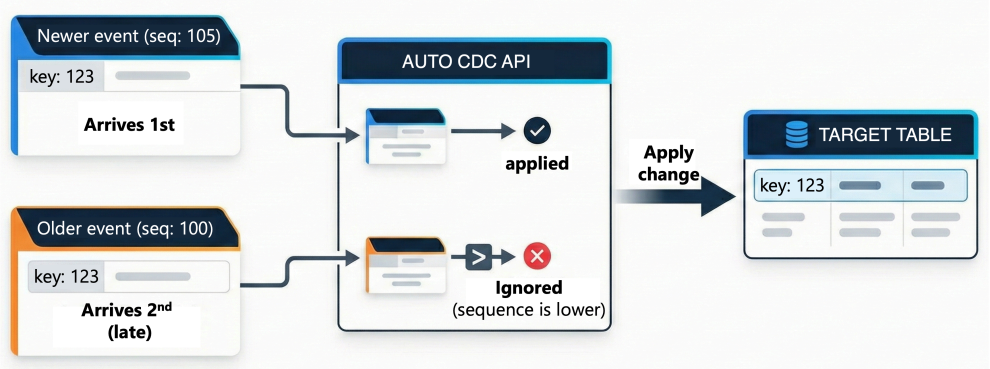
With this configuration, changes to the `last_login` column update the existing record in place. Only changes to other columns—like name or department—create new historical versions.

Handle out-of-order events

Distributed systems often deliver events out of order. A network delay might cause an older update to arrive after a newer one. The AUTO CDC API handles this scenario automatically using the sequence column you specify.

When two updates arrive for the same key, the API compares their sequence values. If the newer update (higher sequence number) has already been applied and an older update arrives late, the API

ignores the late-arriving record. This ensures your destination table always reflects the correct final state.



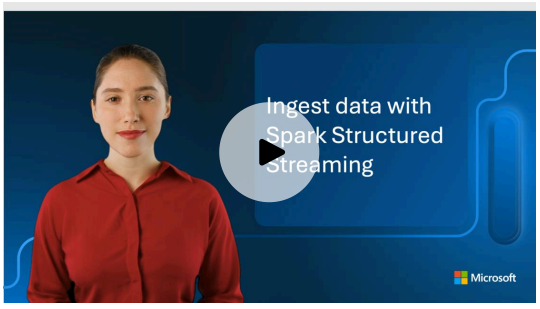
Without this built-in handling, you would need to write complex merge logic that checks timestamps, compares versions, and conditionally applies updates. The AUTO CDC API encapsulates all this logic, letting you focus on your business requirements rather than edge case handling.

6. Ingest data with Spark Structured Streaming

https://learn.microsoft.com/en-us/training/modules/ingest-data-into-unity-catalog/6-ingest-data-spark-structured-streaming

Ingest data with Spark Structured Streaming

Completed



Real-time data ingestion allows you to process events as they occur rather than waiting for batch intervals. When your organization needs to capture IoT sensor readings, clickstream data, or financial transactions as they happen, Spark Structured Streaming provides the foundation for continuous data ingestion into your lakehouse.

In this unit, you learn how to configure Structured Streaming jobs that read from streaming sources and write to Unity Catalog tables.

Understand the Structured Streaming model

Structured Streaming treats incoming data as an unbounded table that grows continuously as new data arrives. You use the same DataFrame API you already know, but instead of `spark.read`, you use `spark.readStream` to create a streaming DataFrame that processes data incrementally.

Consider a scenario where sensor data arrives continuously from manufacturing equipment. Rather than scheduling hourly batch jobs to ingest this data, you can configure a streaming job that processes each reading within seconds of its arrival. The streaming engine tracks which records have been processed using **checkpoints**, so if the job restarts, it resumes from where it left off without reprocessing or missing data.

The fundamental pattern for streaming ingestion involves three steps:

1. **Configure the source** using `spark.readStream` with connection details
2. **Apply transformations** to shape the data as needed
3. **Write to a sink** using `writeStream` with a checkpoint location

Connect to Kafka and Event Hubs

Azure Event Hubs provides an Apache Kafka-compatible endpoint, which means you can use the same Kafka connector for both services. This unified approach simplifies your code when working with different message buses.

The following example reads messages from a Kafka topic:

```
df_stream = (spark.readStream
    .format("kafka")
    .option("kafka.bootstrap.servers", "broker-server:9092")
    .option("subscribe", "sensor-readings")
    .option("startingOffsets", "latest")
    .load())
```

For Azure Event Hubs, you connect through its Kafka-compatible interface. The key difference is the authentication configuration, which uses SASL with your Event Hubs connection string:

```
eh_namespace = "your-namespace"
eh_name = "your-eventhub"
connection_string = dbutils.secrets.get(scope="eventhubs", key="connection-string")

kafka_options = {
    "kafka.bootstrap.servers": f"{eh_namespace}.servicebus.windows.net:9093",
    "subscribe": eh_name,
    "kafka.sasl.mechanism": "PLAIN",
    "kafka.security.protocol": "SASL_SSL",
    "kafka.sasl.jaas.config": f'kafkashaded.org.apache.kafka.common.security.plain
}

df_stream = (spark.readStream
```

```
    .format("kafka")
    .options(**kafka_options)
    .load())
```

When reading from Kafka or Event Hubs through the Kafka-compatible interface, you receive a standard schema with `key`, `value`, `topic`, `partition`, `offset`, and `timestamp` columns. The `key` and `value` columns contain binary data, so you typically cast them to string and parse the content:

```
from pyspark.sql.functions import col, from_json
from pyspark.sql.types import StructType, StringType, DoubleType

schema = StructType() \
    .add("device_id", StringType()) \
    .add("temperature", DoubleType()) \
    .add("timestamp", StringType())

parsed_df = (df_stream
    .select(col("value").cast("string").alias("json_value"))
    .select(from_json(col("json_value"), schema).alias("data"))
    .select("data.*"))
```

Use the native Event Hubs connector

Azure Databricks also provides a native Event Hubs connector that simplifies authentication and provides direct access to Event Hubs-specific features. This connector uses the `eventhubs` format:

```
connection_string = dbutils.secrets.get(scope="eventhubs", key="connection-string")

events = (spark.readStream
    .format("eventhubs")
    .option("eventhubs.connectionString", connection_string)
    .load())
```

The native Event Hubs connector returns a different schema than the Kafka-compatible interface. The event payload is stored in the `body` column (instead of `value`), along with additional Event Hubs-specific metadata columns:

```
# Extract and parse the event payload
payload = events.selectExpr("CAST(body AS STRING) AS payload")
```

You can configure additional Event Hubs options such as consumer group and starting position:

```
events = (spark.readStream
    .format("eventhubs")
    .option("eventhubs.connectionString", connection_string)
    .option("eventhubs.consumerGroup", "$Default")
    .option("eventhubs.startingPosition", '{"offset": "-1", "seqNo": -1, "enqueued
    .load())
```

The `startingPosition` option controls where to begin reading events. Use `-1` for the latest events or configure a specific offset, sequence number, or enqueued time for your ingestion needs.

Configure checkpoint locations

Checkpoints are the mechanism that makes Structured Streaming fault-tolerant. The checkpoint directory stores the offset information that tracks exactly which records have been processed. If your streaming job stops unexpectedly, it can restart and continue from the last committed checkpoint.

> **Important**
>
> Each streaming query must have its own unique checkpoint location. Sharing checkpoints between different queries causes data processing errors.

When you start a streaming write, specify the checkpoint location:

```
checkpoint_path = "/checkpoints/sensor-ingestion"

query = (parsed_df.writeStream
    .format("delta")
    .option("checkpointLocation", checkpoint_path)
    .toTable("catalog.schema.sensor_readings"))
```

Store your checkpoints in a reliable, durable location like Azure Data Lake Storage. The checkpoint files are small but critical for recovery, so configure your storage location with appropriate backup and retention policies.

Write streaming data to Unity Catalog tables

Writing to Unity Catalog tables uses Delta Lake format, which supports both streaming writes and ACID transactions. The `writeStream` API mirrors the batch `write` API but includes options for continuous processing:

```
(parsed_df.writeStream
    .format("delta")
    .outputMode("append")
    .option("checkpointLocation", "/checkpoints/my-stream")
    .toTable("bronze.iot.sensor_readings"))
```

The **output mode** determines how results are written:

- **append**: New rows are added to the table. Use this mode for most ingestion workloads.
- **update**: Changed rows are updated. Use with aggregations that produce updates.
- **complete**: All rows are rewritten. Use with aggregations that require complete recalculation.

For ingestion scenarios where you're appending raw data, the default `append` mode is appropriate.

Control processing with triggers

Triggers determine when and how frequently the streaming engine processes data. Azure Databricks supports several trigger options:

Trigger	Behavior
Default (no trigger)	Processes the next batch as soon as the previous batch completes
`processingTime="10 seconds"`	Processes in fixed intervals
`availableNow=True`	Processes all available data, then stops
`once=True`	Processes one micro-batch, then stops (deprecated)

For continuous ingestion that runs indefinitely, omit the trigger option or use a processing time interval. For incremental batch patterns where you want to process all available data on a schedule, use `availableNow`:

```
(parsed_df.writeStream
    .format("delta")
    .option("checkpointLocation", checkpoint_path)
    .trigger(availableNow=True)
    .toTable("catalog.schema.table_name"))
```

The `availableNow` trigger processes all unprocessed data since the last checkpoint, then terminates. You can schedule this pattern using Databricks Jobs to run at regular intervals, combining the efficiency of streaming (only processing new data) with the predictability of scheduled batch jobs.

Monitor streaming queries

Active streaming queries appear in the Spark UI under the **Structured Streaming** tab. You can also access query status programmatically:

```
query = parsed_df.writeStream.format("delta").option("checkpointLocation", checkpo

# Check if the query is still running
print(query.isActive)

# Get the current status
print(query.status)

# View recent progress
print(query.recentProgress)
```

Use the `awaitTermination()` method to block execution until the stream completes or fails. For production workloads, implement error handling to capture and respond to streaming failures:

```
try:
    query.awaitTermination()
except Exception as e:
    print(f"Stream failed: {e}")
```

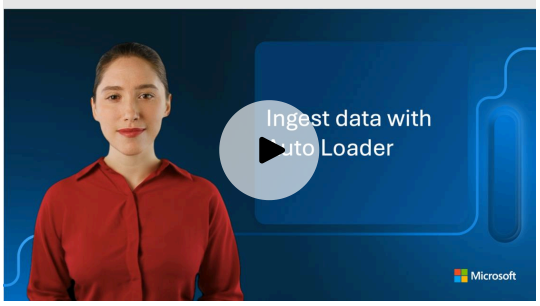
In production environments, configure monitoring and alerts to detect when streaming jobs fall behind or encounter errors. Streaming metrics include lag information that helps you understand whether your processing keeps pace with incoming data.

7. Ingest data with Auto Loader

https://learn.microsoft.com/en-us/training/modules/ingest-data-into-unity-catalog/7-ingest-data-auto-loader

Ingest data with Auto Loader

Completed

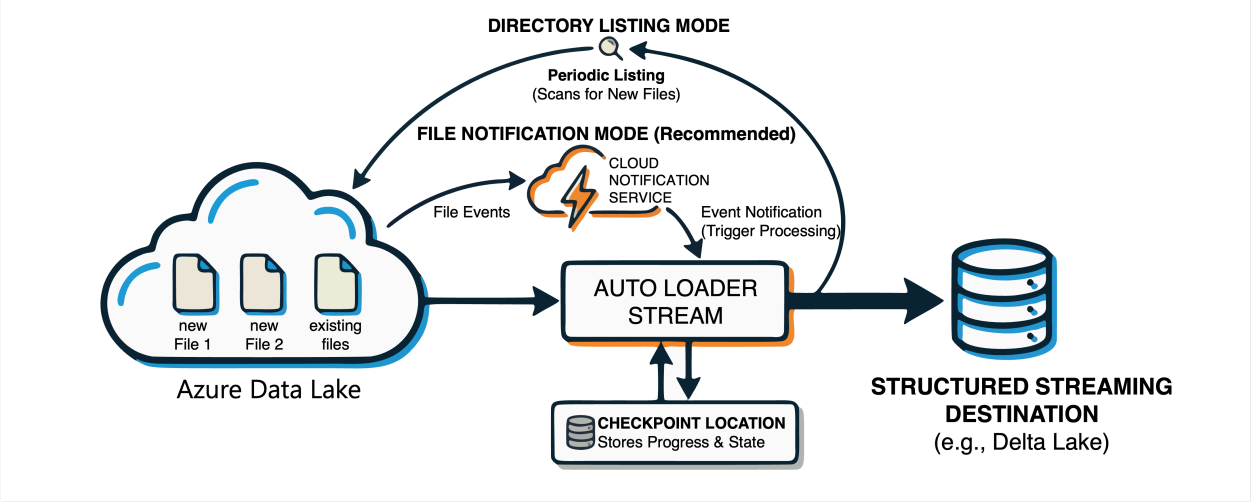


When new data files arrive continuously in cloud storage, you need an efficient way to process them without manually tracking which files have been ingested. **Auto Loader** solves this challenge by automatically detecting and ingesting new files as they appear, providing exactly-once guarantees without requiring you to manage state or checkpoints manually.

Understand how Auto Loader works

Auto Loader is a Structured Streaming source that monitors a cloud storage location and incrementally processes new files. Rather than scanning entire directories repeatedly, Auto Loader maintains state about which files it has already processed, ensuring each file is ingested exactly once.

When you start an Auto Loader stream, it can process existing files in the directory and then continuously watch for new arrivals. The stream stores progress information in a checkpoint location, which allows it to resume from exactly where it stopped if interrupted.



Auto Loader detects new files using one of two modes:

- **Directory listing mode**: Auto Loader periodically lists the input directory to discover new files. This approach requires no additional configuration beyond storage access.
- **File notification mode**: Auto Loader uses cloud notification services to receive events when new files arrive. This mode is more efficient for large-scale workloads because it avoids repeated directory listings.

For most production workloads, Databricks recommends file notification mode with file events enabled on your external location in Unity Catalog. With file events, Auto Loader receives notifications directly when files arrive, which reduces latency and cloud API costs.

Configure Auto Loader for file ingestion

To ingest data with Auto Loader, you use the `cloudFiles` format with `spark.readStream`. The following example reads JSON files from Azure Data Lake Storage and writes them to a Unity Catalog table:

```
base_path = "abfss://container@storage.dfs.core.windows.net/autoloader/orders"
schema_path = f"{base_path}/schema"
checkpoint_path = f"{base_path}/checkpoint"

(spark.readStream
  .format("cloudFiles")
  .option("cloudFiles.format", "json")
  .option("cloudFiles.schemaLocation", schema_path)
  .load("abfss://container@storage.dfs.core.windows.net/incoming/orders/")
  .writeStream
  .option("checkpointLocation", checkpoint_path)
  .trigger(availableNow=True)
  .toTable("sales.bronze.orders"))
```

The `cloudFiles.format` option specifies the source file format. Auto Loader supports JSON, CSV, Parquet, Avro, ORC, XML, text, and binary files. The `cloudFiles.schemaLocation` stores the

inferred schema, which enables Auto Loader to track schema changes over time. While the schema location and checkpoint location can be the same, keeping them separate allows you to reset a stream without losing the inferred schema.

The `trigger(availableNow=True)` setting processes all currently available files and then stops. This pattern works well for scheduled batch jobs. For continuous processing, you can omit the trigger or use `trigger(processingTime="1 minute")` to process files in intervals.

Use SQL syntax with read_files

Auto Loader also supports SQL syntax through the `read_files` table-valued function. This approach is useful when you prefer SQL or when working in SQL notebooks:

```
SELECT * FROM read_files(
  'abfss://container@storage.dfs.core.windows.net/incoming/orders/',
  format => 'json',
  schemaHints => 'order_id INT, amount DECIMAL(10,2)'
)
```

When using `read_files` with streaming tables in Lakeflow Spark Declarative Pipelines, Auto Loader capabilities are automatically enabled. Use the `STREAM` keyword to enable incremental processing:

```
CREATE OR REFRESH STREAMING TABLE bronze_orders
AS SELECT * FROM STREAM read_files(
  'abfss://container@storage.dfs.core.windows.net/incoming/orders/',
  format => 'json'
)
```

In this context, the pipeline manages checkpointing and schema evolution automatically.

> **Tip**
>
> When working with Unity Catalog, store your checkpoint and schema locations in a managed storage location. This ensures consistent governance and avoids permission issues with nested paths.

Handle schema inference and evolution

Auto Loader can automatically detect the schema of your source files, eliminating the need to define schemas manually. When schema inference is enabled, Auto Loader samples files to determine column names and data types.

For JSON and CSV files, Auto Loader infers all columns as strings by default. This conservative approach prevents type mismatches that could cause data loss. To enable automatic type detection, set the `cloudFiles.inferColumnTypes` option:

```
(spark.readStream
  .format("cloudFiles")
  .option("cloudFiles.format", "json")
  .option("cloudFiles.inferColumnTypes", "true")
  .option("cloudFiles.schemaLocation", checkpoint_path)
  .load(source_path))
```

When your source data introduces new columns, Auto Loader detects the change and can evolve the target schema automatically. The `cloudFiles.schemaEvolutionMode` option controls this behavior:

Mode	Behavior
`addNewColumns`	Stream fails. New columns are added to the schema. On restart, stream continues with the updated schema. This is the default when no schema is provided.
`rescue`	Schema is never evolved and stream does not fail. All new columns are captured in the `_rescued_data` column.
`failOnNewColumns`	Stream fails. Stream does not restart unless the provided schema is updated or the offending data file is removed.
`none`	Schema is not evolved, new columns are ignored, and data is not rescued unless the `rescuedDataColumn` option is set. This is the default when a schema is provided.

The **rescued data column** (`_rescued_data`) captures any data that doesn't match the expected schema, including type mismatches and unexpected columns. This feature prevents data loss when source data doesn't conform to expectations. Auto Loader automatically adds this column when inferring the schema.

```
(spark.readStream
  .format("cloudFiles")
  .option("cloudFiles.format", "csv")
  .option("cloudFiles.schemaEvolutionMode", "rescue")
  .schema(expected_schema)
  .load(source_path))
```

To rename the rescued data column, use the `rescuedDataColumn` option:

```
.option("rescuedDataColumn", "my_rescued_data")
```

If you know certain columns should have specific types, use **schema hints** to override the inferred types:

```
.option("cloudFiles.schemaHints", "order_date DATE, amount DECIMAL(10,2)")
```

Monitor and optimize Auto Loader streams

For production workloads, monitoring helps you track ingestion progress and identify issues. Auto Loader provides a SQL function to query the state of discovered files:

```
SELECT * FROM cloud_files_state('/path/to/checkpoint');
```

This query returns metadata about files that Auto Loader has discovered, including their processing status. You can use this information to verify that files are being ingested as expected.

Auto Loader reports metrics through the Structured Streaming progress dashboard, including:

- `numFilesOutstanding`: Files discovered but not yet processed
- `numBytesOutstanding`: Total bytes waiting to be processed

For high-volume workloads, control the processing rate using `cloudFiles.maxFilesPerTrigger` or `cloudFiles.maxBytesPerTrigger`. These settings prevent any single micro-batch from becoming too large:

```
.option("cloudFiles.maxFilesPerTrigger", "1000")
.option("cloudFiles.maxBytesPerTrigger", "1g")
```

> **Important**
>
> The checkpoint location stores critical state information. Avoid applying cloud storage lifecycle policies that might delete checkpoint files, as this corrupts the stream state and requires restarting from scratch.

To access file-level metadata in your ingested data, select the `_metadata` column. This hidden column contains information about each source file, including `file_path`, `file_name`, `file_size`, and `file_modification_time`:

```
(spark.readStream
  .format("cloudFiles")
  .option("cloudFiles.format", "parquet")
  .load(source_path)
  .select("*", "_metadata.file_path", "_metadata.file_modification_time"))
```

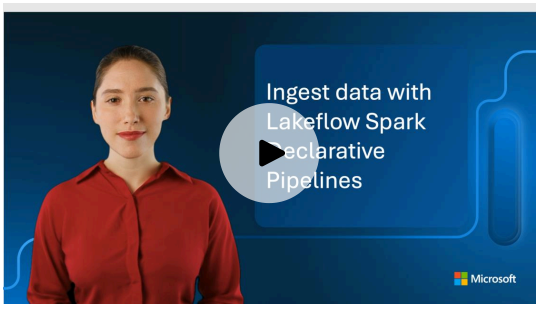
Auto Loader integrates seamlessly with Lakeflow Spark Declarative Pipelines, where it handles checkpointing and schema management automatically. In the next unit, you learn how to use Auto Loader within declarative pipeline definitions for end-to-end data ingestion workflows.

8. Ingest data with Lakeflow Spark Declarative Pipelines

https://learn.microsoft.com/en-us/training/modules/ingest-data-into-unity-catalog/8-ingest-data-lakeflow-declarative-pipelines

Ingest data with Lakeflow Spark Declarative Pipelines

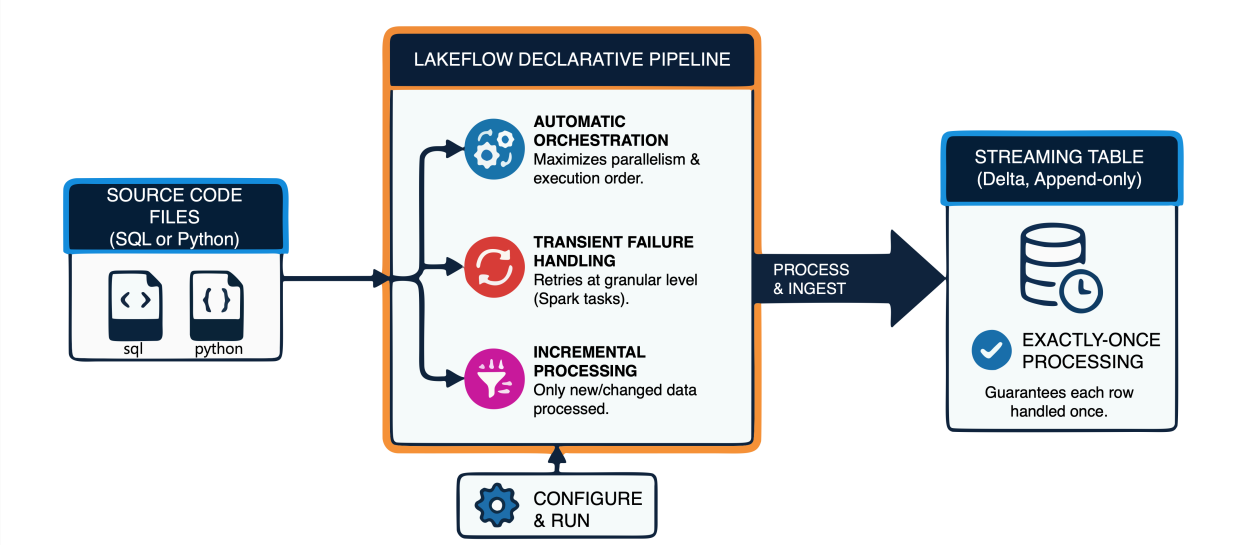
Completed



When you need to ingest data from multiple sources with automated orchestration, schema management, and exactly-once processing guarantees, Lakeflow Spark Declarative Pipelines provides a declarative framework for the entire process. Rather than writing complex imperative code to manage streaming state and dependencies, you define what you want, and the pipeline handles how to execute it.

Understand Lakeflow Spark Declarative Pipelines for ingestion

Lakeflow Spark Declarative Pipelines is a framework for building batch and streaming data pipelines using SQL or Python. You declare the structure of your data transformations, and the pipeline automatically handles orchestration, retries, and incremental processing.



For data ingestion, you typically create **streaming tables** as targets. A streaming table is a Delta table with built-in support for streaming data. Each row from the source is processed exactly once, making streaming tables ideal for append-only ingestion workloads where data continuously arrives.

The pipeline framework provides several benefits for ingestion:

- **Automatic orchestration**: The pipeline determines the correct execution order and maximizes parallelism.
- **Transient failure handling**: If a task fails, the pipeline retries at the most granular level possible, starting with individual Spark tasks.
- **Exactly-once processing**: Streaming tables guarantee that each input row is handled only once.
- **Incremental processing**: Only new or changed data is processed during each update.

To use pipelines, you define streaming tables and flows in source code files (SQL or Python), then configure and run a pipeline that processes these definitions.

Create streaming tables for ingestion

A streaming table acts as the target for your ingestion flow. You define what data to read and how to transform it, and the pipeline manages the execution.

The following SQL example creates a streaming table that reads JSON files from cloud storage:

```
CREATE OR REFRESH STREAMING TABLE orders_bronze
AS SELECT *
FROM STREAM read_files(
  'abfss://container@storage.dfs.core.windows.net/incoming/orders/',
  format => 'json'
)
```

The `STREAM` keyword indicates that the source should be processed incrementally. The `read_files` function reads files from the specified path with automatic format detection. When used with `STREAM`, it enables Auto Loader capabilities for incremental file detection.

In Python, you use the `@dp.table` decorator to define a streaming table:

```
from pyspark import pipelines as dp

@dp.table
def orders_bronze():
    return (
        spark.readStream.format("cloudFiles")
            .option("cloudFiles.format", "json")
            .load("abfss://container@storage.dfs.core.windows.net/incoming/orders/"
    )
```

Both examples create a streaming table named `orders_bronze`. When the pipeline runs, it processes any new files that arrive in the source location. The pipeline manages checkpoints internally, so you don't need to configure checkpoint locations manually.

> **Tip**
>
> Store your pipeline source code in a Git folder for version control. The Lakeflow Pipelines Editor supports a default folder structure with `transformations` for source code and `explorations` for ad hoc analysis.

Configure flows for multiple sources

A **flow** defines how data moves from a source to a target. When you create a streaming table with a query, a default flow is created automatically. However, you can also define flows explicitly to handle more complex scenarios, such as ingesting from multiple sources into a single table.

Consider a scenario where you receive order data from different regional systems. You want to combine them into a single `orders_us` table without using a `UNION` clause. Explicit flows enable this pattern:

```
-- Create the target streaming table
CREATE OR REFRESH STREAMING TABLE orders_us;

-- Add a flow from the west region
CREATE FLOW orders_west_flow
AS INSERT INTO orders_us BY NAME
SELECT * FROM STREAM(orders_west);

-- Add a flow from the east region
CREATE FLOW orders_east_flow
AS INSERT INTO orders_us BY NAME
SELECT * FROM STREAM(orders_east);
```

Each flow appends data to the same target table independently. This approach is more flexible than `UNION` because you can add new source flows without modifying existing ones or triggering a full refresh.

In Python, you use the `@dp.append_flow` decorator:

```
from pyspark import pipelines as dp

# Create the target streaming table
dp.create_streaming_table("orders_us")

# Add flow from west region
@dp.append_flow(target="orders_us")
def orders_west_flow():
    return spark.readStream.table("orders_west")
```

```
# Add flow from east region
@dp.append_flow(target="orders_us")
def orders_east_flow():
    return spark.readStream.table("orders_east")
```

Important

Flow names are used to identify streaming checkpoints. If you rename a flow, the checkpoint doesn't carry over, and the renamed flow starts processing from the beginning as a new flow.

Ingest from message buses

Lakeflow Spark Declarative Pipelines supports ingestion from message buses like Apache Kafka, Azure Event Hubs, Amazon Kinesis, and Google Pub/Sub. For Azure environments, Event Hubs is commonly used and provides a Kafka-compatible endpoint.

The following example reads from a Kafka topic:

```
CREATE OR REFRESH STREAMING TABLE events_raw
AS SELECT *
FROM STREAM read_kafka(
  bootstrapServers => 'kafka_server:9092',
  subscribe => 'events_topic'
)
```

For Azure Event Hubs, you use the Kafka-compatible interface. Event Hubs requires authentication using a connection string stored securely in Azure Databricks secrets:

```
from pyspark import pipelines as dp

EH_NAMESPACE = spark.conf.get("eh.namespace")
EH_NAME = spark.conf.get("eh.name")
EH_CONN_STR = dbutils.secrets.get(scope="eh-secrets", key="connection-string")

KAFKA_OPTIONS = {
    "kafka.bootstrap.servers": f"{EH_NAMESPACE}.servicebus.windows.net:9093",
    "subscribe": EH_NAME,
    "kafka.sasl.mechanism": "PLAIN",
    "kafka.security.protocol": "SASL_SSL",
    "kafka.sasl.jaas.config": f'kafkashaded.org.apache.kafka.common.security.plain
}

@dp.table
def events_bronze():
    return (
        spark.readStream
            .format("kafka")
            .options(**KAFKA_OPTIONS)
```

```
            .load()
    )
```

Store sensitive connection information in Azure Databricks secrets rather than hardcoding values in your pipeline code. Use pipeline parameters (configured in settings) to pass non-sensitive configuration values like namespace names.

Handle schema inference and evolution

When ingesting semi-structured data like JSON, you often don't know the exact schema in advance. Lakeflow Spark Declarative Pipelines provides automatic schema inference and evolution through the `from_json` function.

```
CREATE STREAMING TABLE events_parsed AS
SELECT
    from_json(value, NULL,
        map('schemaLocationKey', 'events_schema',
            'schemaEvolutionMode', 'addNewColumns')) AS parsed
FROM STREAM read_kafka(
    bootstrapServers => 'kafka_server:9092',
    subscribe => 'events_topic'
)
```

Setting the schema to `NULL` enables automatic inference. The `schemaLocationKey` uniquely identifies where the schema is stored. When new fields appear in the source data, the pipeline detects them and can automatically add them to the schema.

The `schemaEvolutionMode` controls how the pipeline responds to schema changes:

Mode	Behavior
`addNewColumns`	New columns are added to the schema automatically. This is the default.
`rescue`	New columns are captured in a `_rescued_data` column instead of being added.
`failOnNewColumns`	The pipeline fails when new columns are detected.
`none`	New columns are ignored and not rescued.

For columns that don't match the inferred schema (type mismatches or unexpected fields), a `_rescued_data` column captures the data to prevent loss.

> **Note**
>
> If you need strict control over the schema, provide explicit schema hints instead of relying on automatic inference: `map('schemaHints', 'order_id INT, amount DECIMAL(10,2)')`

When your ingestion pipeline is running, you can monitor progress through the pipeline event log and the Lakeflow Pipelines Editor. The editor provides execution insights, data previews, and a directed acyclic graph (DAG) showing table dependencies. For production workloads, configure notifications in the pipeline settings to receive alerts when issues occur.

9. Exercise - Ingest Data into Unity Catalog

https://learn.microsoft.com/en-us/training/modules/ingest-data-into-unity-catalog/9-exercise-ingest-data-into-unity-catalog

Exercise - Ingest Data into Unity Catalog

Completed

Now it's your chance to ingest data into Unity Catalog. In this lab, you practise the core data ingestion techniques available in Azure Databricks. You load CSV files from a Unity Catalog managed volume into Delta tables using PySpark DataFrames, `SQL COPY INTO`, and `CREATE TABLE AS SELECT`. You also configure Auto Loader to automatically detect and process new files from cloud storage, demonstrating exactly-once ingestion for continuously arriving data.

> **Note**
>
> To complete this lab, you need an Azure subscription in which you have administrative access.

Launch the exercise and follow the instructions.

Launch Exercise

10. Module assessment

https://learn.microsoft.com/en-us/training/modules/ingest-data-into-unity-catalog/10-knowledge-check

Module assessment

Completed

11. Summary

https://learn.microsoft.com/en-us/training/modules/ingest-data-into-unity-catalog/11-summary

Summary

Completed

Throughout this module, you explored the comprehensive set of data ingestion techniques available in Azure Databricks for loading data into **Unity Catalog**. From managed connectors to custom notebook code, from declarative SQL commands to real-time streaming, each approach addresses specific data patterns and organizational requirements.

You learned how **Lakeflow Connect** simplifies ingestion from enterprise sources by providing managed connectors with built-in change data capture and SCD support. You explored **notebook-based ingestion** using DataFrames and the Spark Structured Streaming API for full control over your ingestion logic. You discovered how **SQL commands** like `COPY INTO` and `CREATE TABLE AS SELECT` provide declarative options for file-based batch loads with automatic file tracking.

For incremental processing, you implemented **CDC flows** using the AUTO CDC API to efficiently apply inserts, updates, and deletes to destination tables. You configured **Spark Structured Streaming** to process events from Kafka and Event Hubs in real time with exactly-once guarantees. You set up **Auto Loader** to automatically detect and ingest new files with schema inference and evolution capabilities. Finally, you used **Lakeflow Spark Declarative Pipelines** to orchestrate end-to-end ingestion workflows with automatic orchestration and error handling.

As you build data pipelines for your organization, consider which ingestion method best matches each source system's characteristics. Use managed connectors when available for common enterprise sources. Choose notebooks for complex transformations or sources that require custom logic. Apply Auto Loader for file-based streaming with automatic schema handling. Orchestrate your ingestion flows with Lakeflow Spark Declarative Pipelines to benefit from built-in reliability features. With these techniques, you can build robust ingestion pipelines that deliver high-quality data to your lakehouse efficiently and reliably.

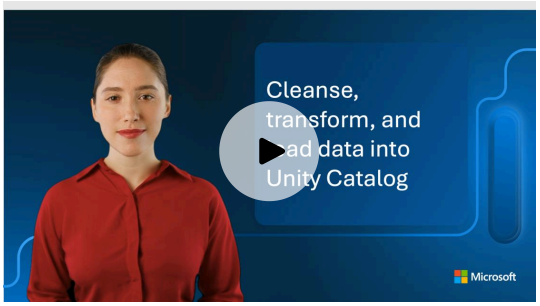
Cleanse, transform, and load data into Unity Catalog

1. Introduction

https://learn.microsoft.com/en-us/training/modules/cleanse-transform-load-data-into-unity-catalog/1-introduction

Introduction

Completed



Raw data rarely arrives in a format ready for analysis. Missing values, duplicate records, inconsistent data types, and poorly structured datasets are common challenges that data engineers must address before data can deliver business value. Without proper cleansing and transformation, analytics results become unreliable and downstream processes fail. **Unity Catalog** in Azure Databricks provides the governance foundation for managing transformed data assets throughout this process.

Azure Databricks offers a comprehensive toolkit for data quality and transformation work. **Data profiling** generates summary statistics that reveal issues like null percentages, unexpected value distributions, and data drift. **Type selection** ensures columns use appropriate data types for storage efficiency and query performance. **Duplicate and null handling** techniques let you identify and resolve common data quality problems using SQL or PySpark. **Transformation operations** including filtering, grouping, aggregation, joins, and set operators reshape data to meet analytical requirements.

Beyond basic transformations, you'll work with advanced reshaping techniques. **Denormalization** flattens related tables for faster query performance. **Pivoting** rotates row values into columns for cross-tabular analysis, while **unpivoting** reverses this process for normalized data structures. Finally, **loading strategies** like append, overwrite, and merge ensure transformed data lands correctly in

target tables—whether you're adding new records, replacing existing data, or synchronizing changes through upsert operations.

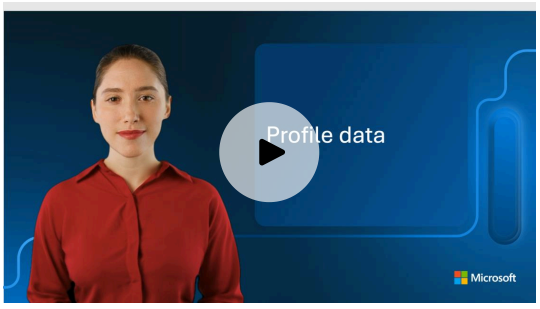
By the end of this module, you'll understand how to assess data quality through profiling, apply cleansing techniques to resolve common issues, transform data using SQL and PySpark operations, and load the results into Unity Catalog tables with the appropriate strategy for each scenario.

2. Profile data

https://learn.microsoft.com/en-us/training/modules/cleanse-transform-load-data-into-unity-catalog/2-profile-data-summary-statistics

Profile data

Completed

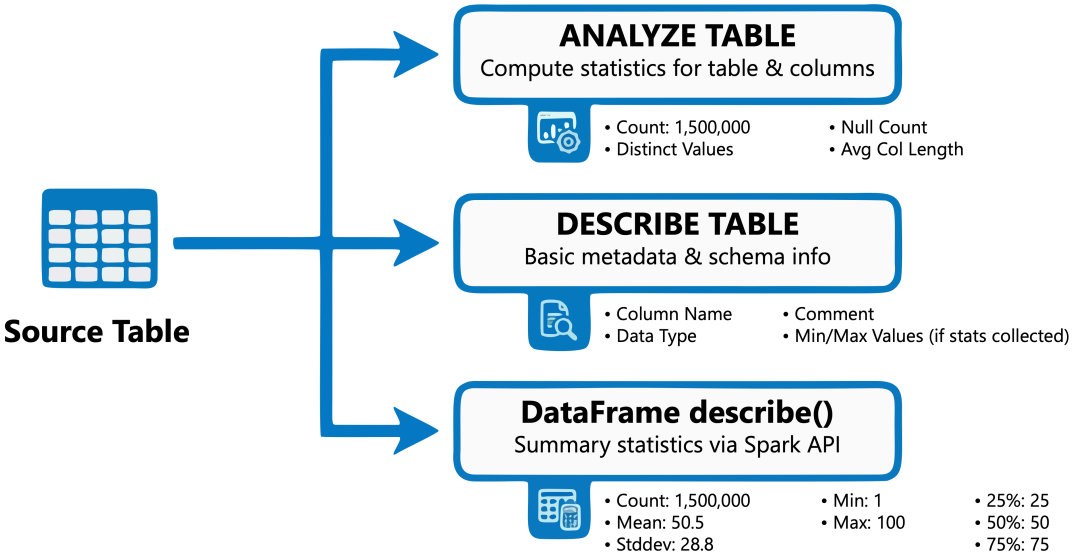


Before you cleanse or transform data, you need to understand what's in your tables. **Profiling data** means examining your datasets to generate summary statistics—counts, averages, value distributions, and null percentages—that reveal the quality and structure of your data. This foundation helps you identify issues like missing values, outliers, and unexpected patterns before they cause problems downstream.

In this unit, you learn how to generate summary statistics using SQL commands and the data profiling feature in Azure Databricks.

Generate summary statistics using SQL

Azure Databricks provides SQL commands that collect and display statistics about your tables. These statistics help the query optimizer choose efficient execution plans and give you insight into your data's characteristics.



Collect statistics with ANALYZE TABLE

The `ANALYZE TABLE` command computes statistics for a table or specific columns. These statistics include row counts, column-level metrics like minimum and maximum values, null counts, and distinct value counts.

```
-- Collect basic table statistics (row count and size)
ANALYZE TABLE sales.transactions COMPUTE STATISTICS;

-- Collect statistics for specific columns
ANALYZE TABLE sales.transactions COMPUTE STATISTICS FOR COLUMNS
    transaction_id, amount, customer_id;

-- Collect statistics for all columns
ANALYZE TABLE sales.transactions COMPUTE STATISTICS FOR ALL COLUMNS;
```

When you analyze a specific column, you get detailed metrics:

```
-- View column-level statistics
DESC EXTENDED sales.transactions amount;
```

This returns metrics including minimum value, maximum value, number of nulls, distinct count, average column length, and maximum column length.

View statistics with DESCRIBE TABLE

The `DESCRIBE TABLE EXTENDED` command displays collected statistics alongside table metadata:

```
DESCRIBE TABLE EXTENDED sales.transactions;
```

You can also use PySpark to retrieve table metadata and generate summary statistics:

```
# Generate summary statistics for numeric columns using the DataFrame API
df = spark.table("sales.transactions")
df.describe().show()
```

The `describe()` method returns count, mean, standard deviation, min, and max for numeric columns.

The output shows the table's size in bytes and row count, confirming that profiling has been performed. For Unity Catalog managed tables, predictive optimization automatically runs `ANALYZE` to keep statistics current.

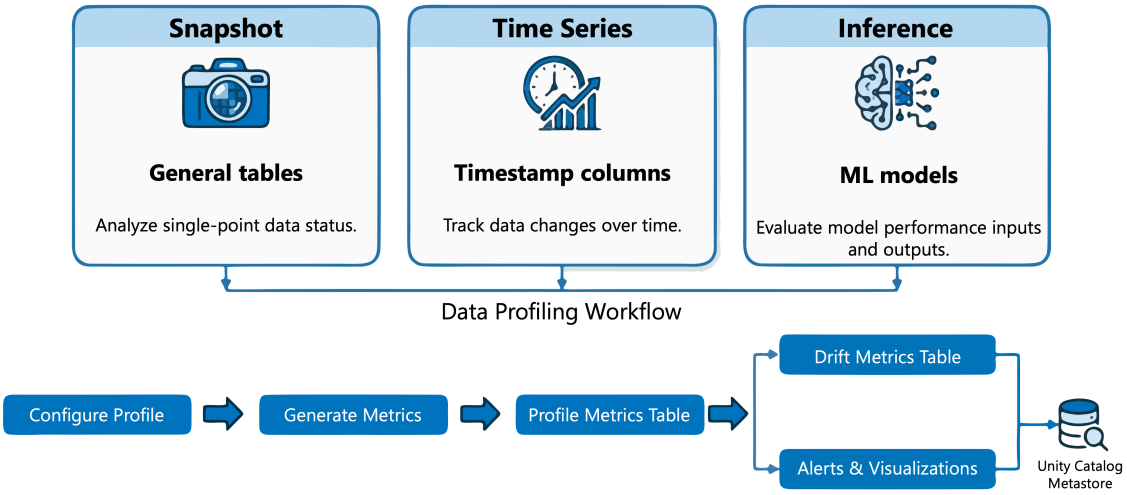
> **Tip**
>
> Run `ANALYZE TABLE` after bulk data loads or significant changes to ensure the query optimizer has accurate statistics for planning efficient queries.

Use data profiling in Unity Catalog

Beyond SQL commands, Azure Databricks provides **data quality monitoring** in Unity Catalog—an umbrella that covers two distinct capabilities:

- **Anomaly detection** monitors all tables in a schema automatically. It analyzes historical patterns to evaluate each table's freshness (how recently it was updated) and completeness (whether the expected number of rows arrived). You enable it once at the schema level in Catalog Explorer, and Azure Databricks handles the rest using intelligent scanning. Use anomaly detection for broad, automated coverage across all your tables without per-table configuration.
- **Data profiling** provides deep per-table statistical analysis—tracking distributions, null percentages, drift metrics, and model performance over time. You configure it individually for each table. Use data profiling for your most critical tables where you need detailed historical trends.

For data engineering work focused on cleansing and quality, you'll use both: anomaly detection as an early-warning system across all tables, and data profiling for in-depth monitoring of key tables.



Understand profile types

Data profiling supports three analysis types, each suited to different scenarios:

Profile type	Use case	Key characteristics
Snapshot	General-purpose tables	Computes metrics over the entire table with each refresh
Time series	Tables with timestamp columns	Tracks metrics across time windows to detect trends
Inference	ML model inference tables	Monitors model inputs, predictions, and accuracy over time

For data engineering tasks focused on data quality, the snapshot and time series profiles are most relevant.

Create a profile using the UI

To create a profile in Catalog Explorer:

1. Navigate to your table in Catalog Explorer.
2. Select the **Quality** tab.
3. If anomaly detection is **not** enabled for the schema, select **Enable**. If anomaly detection is already enabled, select **Configure**.
4. In the **Data Quality Monitoring** dialog, in the **Data profiling** field, select **Configure**.
5. Choose your profile type and configure options like granularity and scheduling.

The profile generates two metric tables: a profile metrics table with summary statistics and a drift metrics table tracking distribution changes over time.

Create a profile using the API

For programmatic access, use the Databricks SDK. Install or upgrade the SDK to get the current API:

```
%pip install "databricks-sdk>=0.68.0"
```

Then create a snapshot profile for a table:

```
from databricks.sdk import WorkspaceClient
from databricks.sdk.service.dataquality import Monitor, DataProfilingConfig, Snapsh

w = WorkspaceClient()

# Retrieve the table ID and schema ID required by the API
table = w.tables.get(full_name="main.sales.transactions")
schema = w.schemas.get(full_name="main.sales")

w.data_quality.create_monitor(
    monitor=Monitor(
        object_type="table",
        object_id=table.table_id,
        data_profiling_config=DataProfilingConfig(
            output_schema_id=schema.schema_id,
            assets_dir="/Workspace/Users/user@example.com/monitoring",
            snapshot=SnapshotConfig()
        )
    )
)
```

To refresh the profile and update metrics:

```
from databricks.sdk.service.dataquality import Refresh

w.data_quality.create_refresh(
    object_type="table",
    object_id=table.table_id,
    refresh=Refresh(object_type="table", object_id=table.table_id)
)
```

Interpret profiling results

Profiling generates metrics that help you assess data quality and identify issues requiring attention.

Key metrics to examine

The profile metrics table contains statistics for each column:

Metric	What it reveals
`count` and `num_nulls`	Data completeness—high null percentages might indicate ingestion issues
`distinct_count`	Cardinality—unexpected values might suggest data quality problems
`min`, `max`, `avg`, `stddev`	Distribution characteristics—outliers or skewed distributions

Metric	What it reveals
`frequent_items`	Most common values—helps identify dominant categories or duplicates
`quantiles`	Value distribution—useful for detecting skewness

Detect data quality issues

When you examine profiling results, look for these common indicators:

- **High null percentages** in columns that should have values
- **Unexpected distinct counts**, such as unique identifiers that aren't unique
- **Value ranges outside expected bounds**, like negative amounts where only positives are valid
- **Distribution changes over time**, which can indicate data drift

The drift metrics table calculates statistical tests like the Kolmogorov-Smirnov (KS) test for numeric columns and chi-squared test for categorical columns. These tests quantify how much distributions have changed.

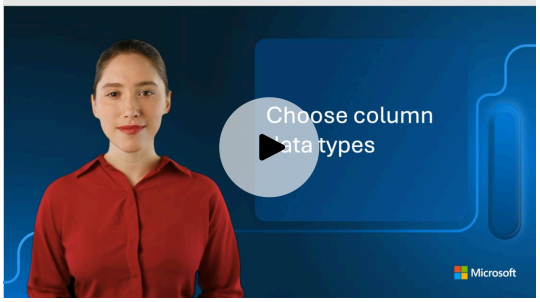
Understanding these patterns helps you design appropriate cleansing and validation rules for your data pipelines.

3. Choose column data types

https://learn.microsoft.com/en-us/training/modules/cleanse-transform-load-data-into-unity-catalog/3-choose-appropriate-column-data-types

Choose column data types

Completed



Choosing the right data type for each column directly affects your data quality, storage efficiency, and query performance. When you select an inappropriate type, you risk losing precision, wasting storage space, or blocking operations that require specific type characteristics. In this unit, you learn how to evaluate your data and select the most appropriate column types in Azure Databricks.

Understand data type categories

Azure Databricks organizes data types into categories based on their characteristics and use cases. Understanding these categories helps you narrow down your options when choosing a type for a specific column.

Numeric types handle mathematical values and calculations. You have several options:

Type category	Types	Use case
Integral	`TINYINT`, `SMALLINT`, `INT`, `BIGINT`	Whole numbers of varying sizes
Decimal	`DECIMAL(p,s)`	Exact precision for financial data
Floating point	`FLOAT`, `DOUBLE`	Scientific calculations with approximation

Date-time types represent temporal data:

Type	Description
`DATE`	Calendar dates without time components
`TIMESTAMP`	Date and time with session timezone
`TIMESTAMP_NTZ`	Date and time without timezone

Text and binary types store character and byte sequences:

- `STRING` for variable-length text
- `BINARY` for raw byte sequences

Complex types organize nested or repeated data:

- `ARRAY` for ordered sequences of elements
- `MAP` for key-value pairs
- `STRUCT` for records with named fields

Select the right numeric type

Numeric data requires careful type selection because different types have different precision, range, and storage characteristics. Your choice affects both data integrity and performance.

For whole numbers, match the type to your value range:

Type	Range	Storage	Common use cases
`TINYINT`	-128 to 127	1 byte	Status codes, small counters
`SMALLINT`	-32,768 to 32,767	2 bytes	Year values, limited ranges
`INT`	-2.1 billion to 2.1 billion	4 bytes	Most identifiers, counts

Type	Range	Storage	Common use cases
`BIGINT`	-9.2 quintillion to 9.2 quintillion	8 bytes	Large IDs, row counts

When creating tables, specify the appropriate type for each column:

```
CREATE TABLE orders (
    order_id INT,
    status_code TINYINT,
    total_items SMALLINT,
    transaction_id BIGINT
);
```

For decimal numbers, the choice between `DECIMAL` and floating point types depends on your precision requirements:

```
-- DECIMAL(p,s): exact precision for financial calculations
-- p = total digits, s = digits after decimal point
CREATE TABLE transactions (
    transaction_id INT,
    amount DECIMAL(10,2),         -- Exact: up to 99,999,999.99
    tax_rate DECIMAL(5,4)         -- Exact: up to 9.9999
);

-- DOUBLE: approximate, suitable for scientific calculations
CREATE TABLE measurements (
    sensor_id INT,
    temperature DOUBLE,           -- Approximate, large range
    pressure DOUBLE
);
```

Important

Use `DECIMAL` for financial data where exact precision matters. Floating point types (`FLOAT` , `DOUBLE`) use binary representation and can introduce rounding errors with base-10 values like currency.

Choose temporal types correctly

Temporal data type selection depends on whether you need timezone awareness and what level of precision your application requires.

Use `DATE` when you only need calendar dates without time components:

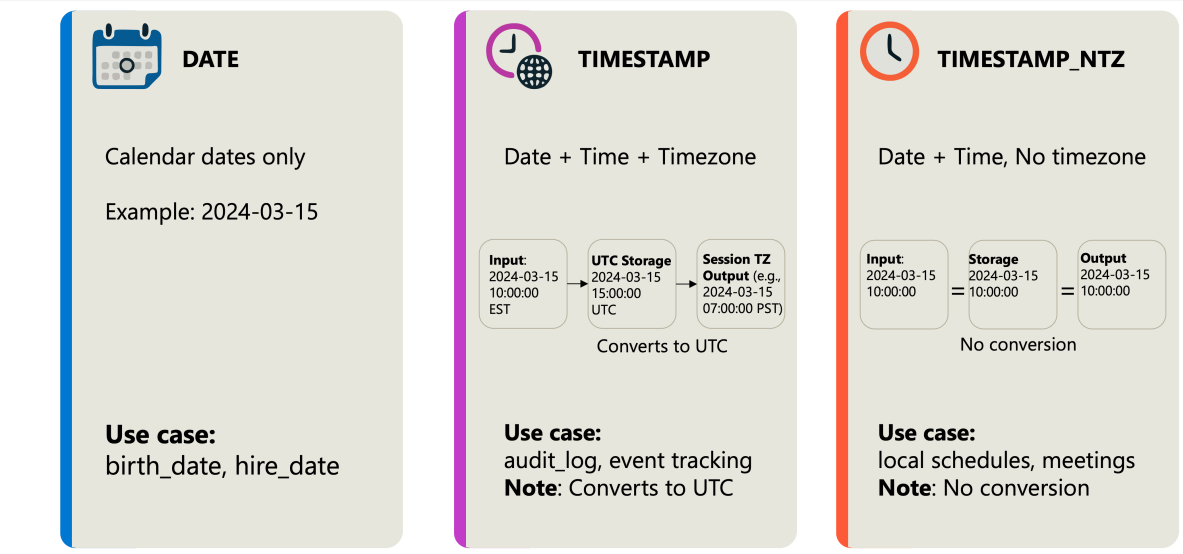
```
CREATE TABLE employees (
    employee_id INT,
    hire_date DATE,
```

```
    birth_date DATE
);
```

For timestamps, choose between timezone-aware and timezone-naive types:

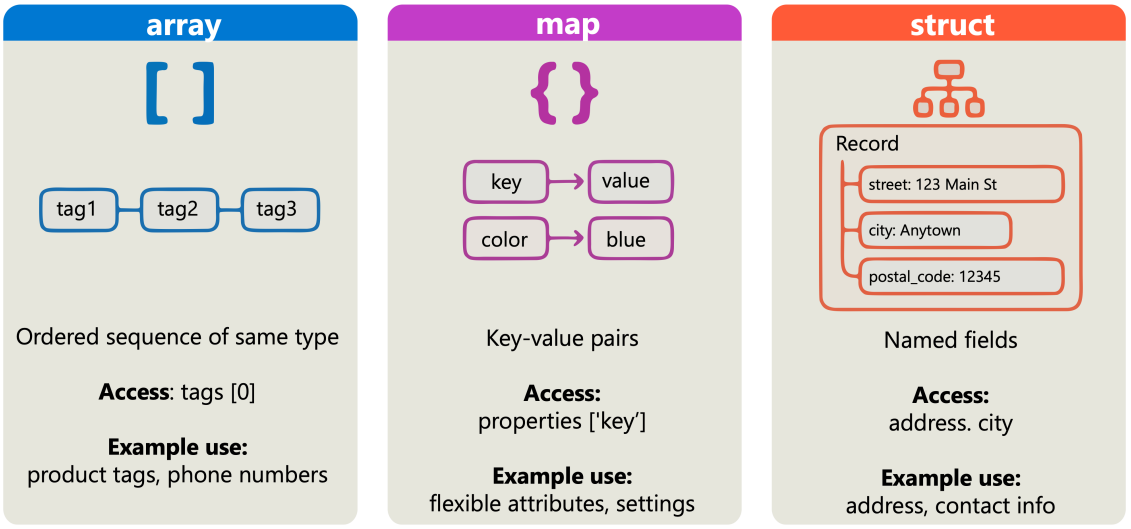
```
-- TIMESTAMP: normalizes to UTC internally, applies session timezone on read
-- Use when tracking events across time zones
CREATE TABLE audit_log (
    log_id BIGINT,
    event_time TIMESTAMP,
    user_id INT
);

-- TIMESTAMP_NTZ: no timezone conversion
-- Use when the time should remain as entered
CREATE TABLE schedules (
    schedule_id INT,
    local_meeting_time TIMESTAMP_NTZ
);
```

The `TIMESTAMP` type converts values to UTC for storage and applies the session timezone when retrieving values. This behavior ensures consistent ordering across time zones but can cause confusion if you expect values to remain unchanged.



Work with complex types

Complex types let you store structured, nested, or repeated data within a single column. They're useful when your data doesn't fit a flat relational model.



Use `ARRAY` when you have multiple values of the same type:

```
-- Store a list of tags for each product
CREATE TABLE products (
    product_id INT,
    name STRING,
    tags ARRAY<STRING>
);

-- Query array elements
SELECT product_id, tags[0] AS primary_tag FROM products;
```

Use `MAP` for key-value pairs with dynamic keys:

```
-- Store flexible attributes as key-value pairs
CREATE TABLE devices (
    device_id INT,
    properties MAP<STRING, STRING>
);

-- Access map values by key
SELECT device_id, properties['firmware_version'] AS firmware FROM devices;
```

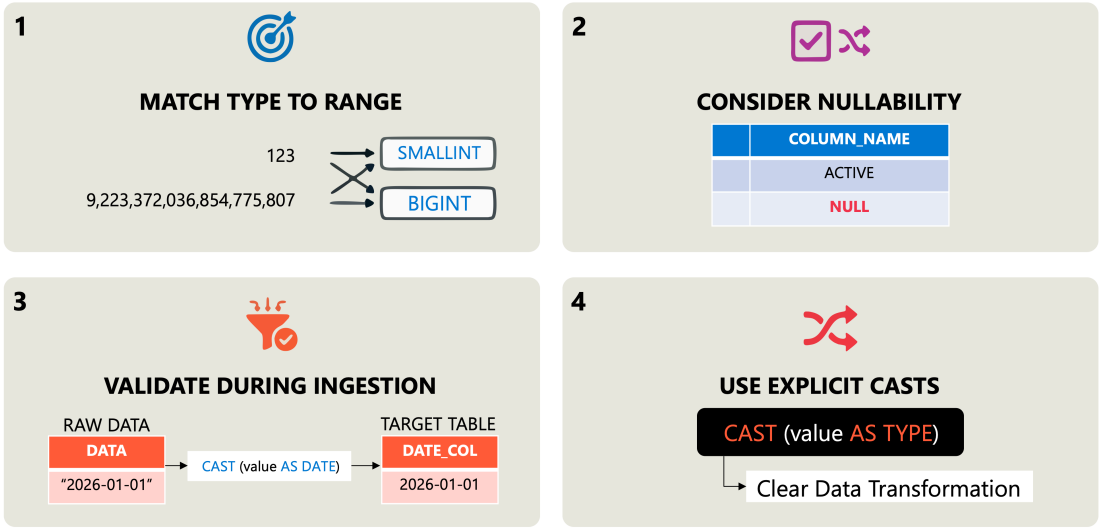
Use `STRUCT` when you have a fixed set of named fields:

```
-- Store address as a structured type
CREATE TABLE customers (
    customer_id INT,
    address STRUCT<street: STRING, city: STRING, postal_code: STRING>
);

-- Access struct fields
SELECT customer_id, address.city FROM customers;
```

> **Tip**
>
> Complex types reduce the need for additional tables and joins, but they also make some operations more difficult. Consider your query patterns before choosing between normalized tables and complex types.

Apply type selection best practices

When evaluating data for type selection, consider these practical guidelines:



Match the type to the data's nature and range. Don't use `BIGINT` for values that never exceed a few thousand. The extra bytes multiply across millions of rows.

Consider nullability. If a column should never contain null values, document this constraint. While data types don't enforce nullability by default, you can add `NOT NULL` constraints to table definitions.

Validate types during ingestion. Cast incoming data to the target type early in your pipeline to catch type mismatches before they propagate:

```
-- Cast during data loading
INSERT INTO target_table
SELECT
    CAST(raw_id AS INT) AS id,
    CAST(raw_amount AS DECIMAL(10,2)) AS amount,
    CAST(raw_date AS DATE) AS transaction_date
FROM staging_table;
```

Use explicit casts when converting types. Implicit conversions can succeed silently even when data is lost. Explicit casts make your intentions clear and help identify problems during testing.

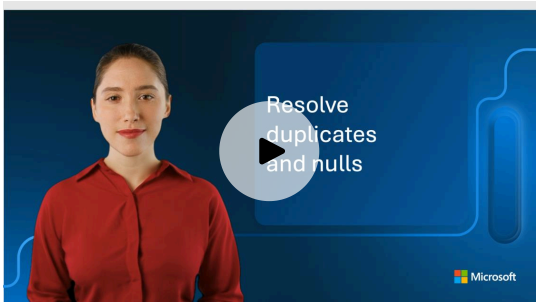
These practices help you build pipelines where data types serve as a form of documentation and validation, catching errors early rather than letting them affect downstream analytics.

4. Resolve duplicates and nulls

https://learn.microsoft.com/en-us/training/modules/cleanse-transform-load-data-into-unity-catalog/4-resolve-duplicate-missing-null-values

Resolve duplicates and nulls

Completed

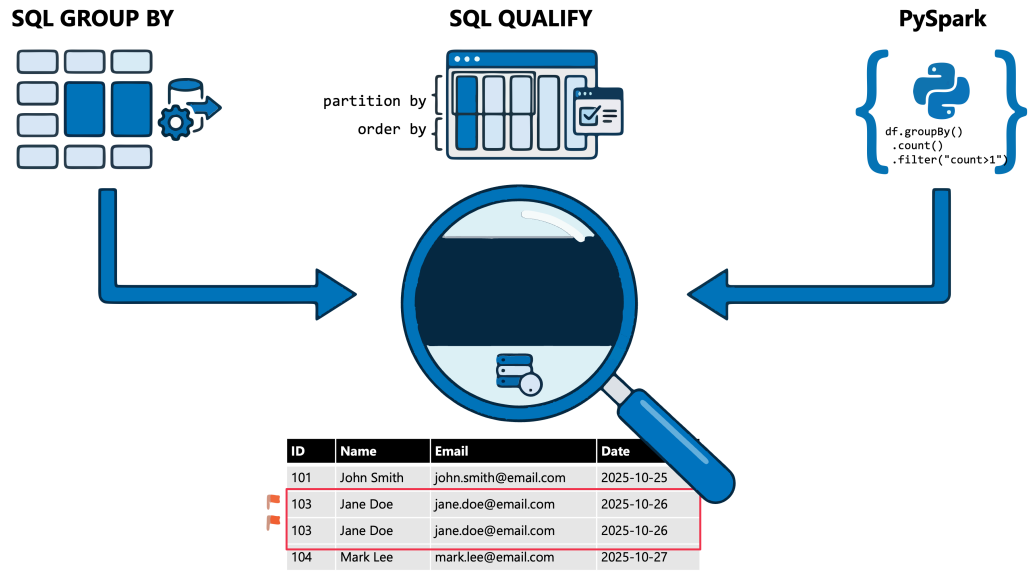


Data quality issues like duplicate records and null values can compromise your analytics and downstream processes. Whether duplicates arise from concurrent data loads or nulls appear due to incomplete source data, addressing these issues is a core responsibility for data engineers working in Azure Databricks.

In this unit, you learn how to identify and resolve duplicate, missing, and null values using both SQL and PySpark approaches.

Identify duplicate records

Before you can remove duplicates, you need to find them. Duplicates typically occur when the same record appears multiple times in a table, often due to data integration issues, repeated ingestion, or concurrent write operations.



Use GROUP BY and HAVING to find duplicates

The most straightforward way to identify duplicates is by grouping records and counting occurrences. The `HAVING` clause filters the grouped results to show only values that appear more than once.

```
SELECT
    customer_id,
    email,
    COUNT(*) AS occurrence_count
FROM sales.customers
GROUP BY customer_id, email
HAVING COUNT(*) > 1
ORDER BY occurrence_count DESC;
```

This query returns all combinations of `customer_id` and `email` that appear multiple times in the table, helping you understand the scope of duplication.

Use QUALIFY with window functions

The `QUALIFY` clause provides a more elegant approach by filtering the results of window functions directly, without requiring subqueries. This approach is particularly useful when you need to see the actual duplicate rows, not just counts.

```
SELECT *
FROM sales.customers
QUALIFY COUNT(*) OVER (PARTITION BY customer_id, email) > 1;
```

This query might return results like:

customer_id	email	name	created_at	status
1001	jane@contoso.com	Jane Doe	2024-01-15	Active
1001	jane@contoso.com	Jane D.	2024-03-20	Active
1002	bob@fabrikam.com	Bob Smith	2024-02-10	Pending
1002	bob@fabrikam.com	Robert Smith	2024-02-11	Active

With this pattern, you get all columns from the duplicate rows, making it easier to investigate the source of duplication.

Find duplicates using PySpark

In PySpark, you can use window functions combined with filtering to identify duplicate records:

```
from pyspark.sql.functions import count
from pyspark.sql.window import Window

window_spec = Window.partitionBy("customer_id", "email")

duplicates_df = (
    df.withColumn("row_count", count("*").over(window_spec))
    .filter("row_count > 1")
    .drop("row_count")
)

display(duplicates_df)
```

Alternatively, use the `exceptAll()` method to compare the original DataFrame against a deduplicated version:

```
df_duplicates = df.exceptAll(df.dropDuplicates(["customer_id", "email"]))
display(df_duplicates)
```

The `exceptAll()` method performs a set difference that preserves duplicates. It returns all rows from the first DataFrame that aren't in the second DataFrame, keeping duplicate occurrences. Here's how it works:

1. `df.dropDuplicates(["customer_id", "email"])` creates a DataFrame with one row per unique combination
2. `exceptAll()` compares the original DataFrame against this deduplicated version
3. For each unique combination, one occurrence is "matched" and removed, leaving only the extra duplicates

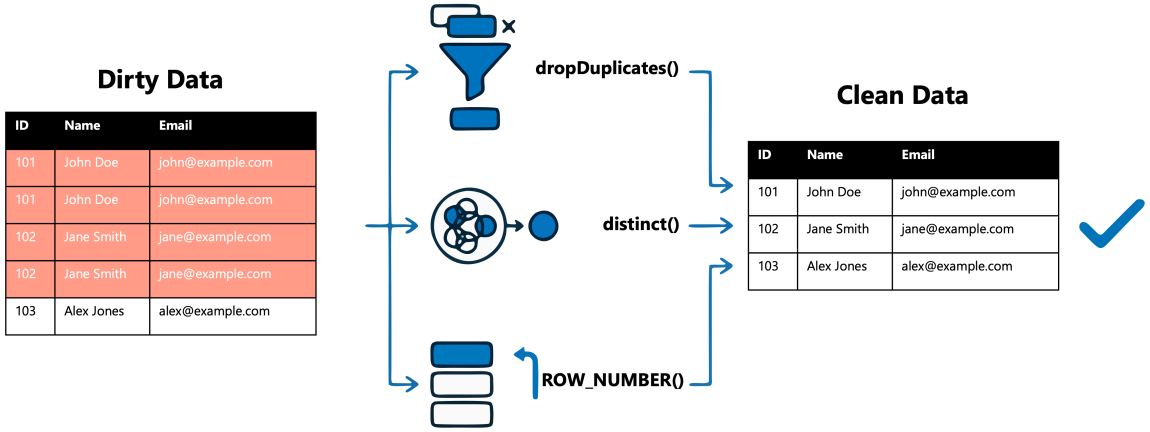
For example, if customer 1001 appears three times in the original DataFrame and once in the deduplicated version, `exceptAll()` returns two rows (the extra occurrences).

> **Note**

> Unlike `except()`, which removes all matching rows and returns distinct results, `exceptAll()` preserves the duplicate structure. If you have three identical rows in the original and one in the deduplicated set, `exceptAll()` returns exactly two rows.

Remove duplicate records

Once you've identified duplicates, you can choose a resolution strategy based on your data requirements.



Remove duplicates in PySpark

The `dropDuplicates()` method removes duplicate rows based on specified columns, keeping only the first occurrence:

```
df_clean = df.dropDuplicates(["customer_id", "email"])
```

For complete row deduplication across all columns, use `distinct()`:

```
df_unique = df.distinct()
```

Keep specific rows using ROW_NUMBER

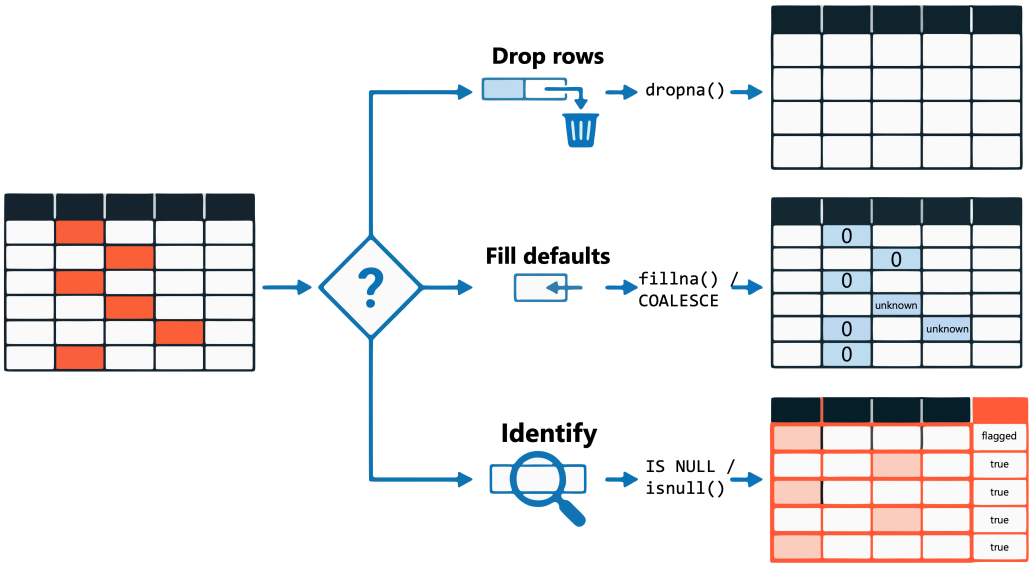
When you need control over which duplicate to keep, use `ROW_NUMBER()` with `QUALIFY` to select records based on criteria like the most recent update:

```
SELECT *
FROM sales.customers
QUALIFY ROW_NUMBER() OVER (
    PARTITION BY customer_id
    ORDER BY updated_at DESC
) = 1;
```

This query partitions records by `customer_id`, orders them by `updated_at` in descending order, and keeps only the most recently updated record for each customer.

Handle null and missing values

Null values represent missing or unknown data. Depending on your analysis requirements, you might need to remove rows with nulls, replace them with default values, or use statistical imputation.



Identify null values

Use the `isnull()` function or `IS NULL` operator to find records with missing values:

```
SELECT *
FROM sales.transactions
WHERE amount IS NULL
   OR customer_id IS NULL;
```

To count null occurrences per column:

```
SELECT
    COUNT(*) AS total_rows,
    COUNT(*) - COUNT(amount) AS null_amount_count,
    COUNT(*) - COUNT(customer_id) AS null_customer_count
FROM sales.transactions;
```

Drop rows with null values

In PySpark, the `dropna()` method removes rows containing null values:

```
# Drop rows where any column is null
df_clean = df.dropna()
```

```
# Drop rows where all columns are null
df_clean = df.dropna(how="all")

# Drop rows where specific columns are null
df_clean = df.dropna(subset=["customer_id", "amount"])
```

The `how` parameter controls the behavior: `"any"` drops rows with at least one null (default), while `"all"` drops rows only when all specified columns are null.

Fill null values with defaults

The `fillna()` method replaces null values with specified defaults:

```
# Fill all null values with a single value
df_filled = df.fillna(0)

# Fill specific columns with different values
df_filled = df.fillna({
    "amount": 0,
    "status": "Unknown",
    "quantity": 1
})
```

In SQL, use the `COALESCE` function to provide default values:

```
SELECT
    customer_id,
    COALESCE(amount, 0) AS amount,
    COALESCE(status, 'Unknown') AS status
FROM sales.transactions;
```

Choose the right strategy

The appropriate null handling strategy depends on your data context:

Scenario	Recommended approach
Nulls in required fields like identifiers	Drop the rows
Nulls in numeric fields for aggregation	Fill with zero or mean
Nulls in optional categorical fields	Fill with placeholder like "Unknown"
Nulls that would skew analysis	Drop or fill based on domain knowledge

> **Tip**
>
> Document your null handling decisions. Future analysts need to understand why certain values were replaced or removed to correctly interpret query results.

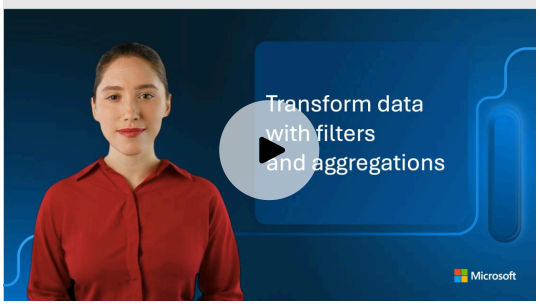
Understanding how to identify and resolve these data quality issues prepares you to build more reliable data pipelines. With clean data, your downstream transformations and analytics produce trustworthy results.

5. Transform data with filters and aggregations

https://learn.microsoft.com/en-us/training/modules/cleanse-transform-load-data-into-unity-catalog/5-transform-data-filter-group-aggregate

Transform data with filters and aggregations

Completed

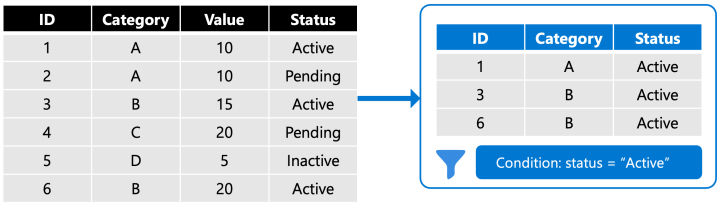


Data transformation turns raw data into formats suitable for analysis and reporting. **Filtering**, **grouping**, and **aggregating** are fundamental operations you use to shape data—selecting relevant records, organizing them into meaningful categories, and calculating summary statistics. These operations form the foundation of most data engineering pipelines.

In this unit, you learn how to filter rows based on conditions, group data by one or more columns, and apply aggregate functions to calculate summary values.

Filter data to select relevant rows

Filtering removes rows that don't meet your criteria, reducing dataset size and focusing on the data that matters. In Azure Databricks, you can filter data using PySpark's DataFrame API or SQL.



Filter with PySpark

The `filter()` and `where()` methods work identically—use whichever you prefer for readability. The `col()` function references column names in your expressions.

```
from pyspark.sql.functions import col

# Filter customers with account balance greater than 1000
df_filtered = df_customer.filter(col("c_acctbal") > 1000)
display(df_filtered)
```

For multiple conditions, combine them with logical operators. Use `&` for AND and `|` for OR, wrapping each condition in parentheses:

```
# Filter customers in nation 20 with balance over 1000
df_filtered = df_customer.filter(
    (col("c_nationkey") == 20) & (col("c_acctbal") > 1000)
)

# Filter customers matching either of two IDs
df_filtered = df_customer.filter(
    (col("c_custkey") == 412446) | (col("c_custkey") == 412447)
)
```

Filter with SQL

SQL's `WHERE` clause provides the same filtering capability with familiar syntax:

```
-- Filter orders with status 'F' and total price above 50000
SELECT *
FROM orders
WHERE o_orderstatus = 'F'
  AND o_totalprice > 50000;
```

You can filter directly when reading from a table in PySpark using `spark.sql()`:

```
df_filtered = spark.sql("""
    SELECT *
    FROM sales.transactions
    WHERE amount > 100
      AND transaction_date >= '2024-01-01'
""")
```

Filter null values

Filtering null values requires special handling in Spark DataFrames. Use the `isNull()` and `isNotNull()` functions to identify or exclude null values:

```
# Filter rows where order_amount is not null
df_valid_orders = df.filter(col("order_amount").isNotNull())
```

```
# Filter rows where order_amount is null
df_null_orders = df.filter(col("order_amount").isNull())

# Alternative syntax using column object directly
df_valid_orders = df.filter(df.order_amount.isNotNull())
```

Important

Using Python's `None` with inequality operators like `!= None` doesn't reliably filter null values in Spark DataFrames. Null comparisons in SQL semantics don't evaluate to true or false—they return null. Always use `isNull()` or `isNotNull()` for correct null handling.

In SQL, use the `IS NULL` or `IS NOT NULL` operators:

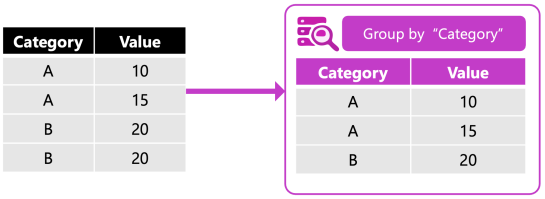
```
-- Filter orders with non-null amounts
SELECT *
FROM orders
WHERE order_amount IS NOT NULL;
```

For comprehensive null handling that removes entire rows containing null values, use the `dropna()` method covered in the unit on resolving duplicate and missing values:

```
# Remove rows where order_amount is null
df_clean = df.dropna(subset=["order_amount"])
```

Group data to organize records

Grouping organizes rows that share common values into categories. This prepares data for aggregation—once grouped, you can calculate statistics for each category.



Group with PySpark

Use the `groupBy()` method to specify one or more columns for grouping:

```
# Group by a single column
df_grouped = df_customer.groupBy("c_mktsegment")

# Group by multiple columns
df_grouped = df_customer.groupBy("c_mktsegment", "c_nationkey")
```

Grouping alone returns a `GroupedData` object. To see results, you must apply an aggregation.

Group with SQL

SQL uses the `GROUP BY` clause to organize rows:

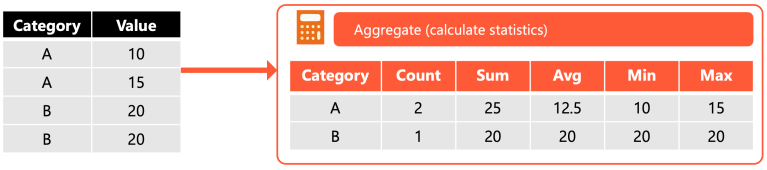
```
-- Group sales by region
SELECT region, COUNT(*) as order_count
FROM sales
GROUP BY region;
```

When grouping, every column in your `SELECT` statement must either appear in the `GROUP BY` clause or be wrapped in an aggregate function.

Aggregate data to calculate summary statistics

Aggregation calculates summary values—counts, sums, averages, and extremes—for each group. Common aggregate functions include:

Function	Description	Example use case
`count()`	Counts rows in each group	Number of orders per customer
`sum()`	Adds values in a column	Total revenue by product
`avg()`	Calculates the mean	Average order value by region
`min()`	Finds the smallest value	Earliest order date per customer
`max()`	Finds the largest value	Highest transaction amount



Aggregate with PySpark

Use the `agg()` method to apply aggregate functions after grouping. Import functions from `pyspark.sql.functions`:

```
from pyspark.sql.functions import avg, sum, count, min, max

# Calculate average balance by market segment
df_segment_avg = df_customer.groupBy("c_mktsegment").agg(
    avg("c_acctbal").alias("avg_balance")
```

```
)
display(df_segment_avg)
```

Apply multiple aggregations in a single statement:

```
# Multiple aggregations per group
df_order_stats = df_order.groupBy("o_orderpriority").agg(
    count("o_orderkey").alias("order_count"),
    sum("o_totalprice").alias("total_revenue"),
    avg("o_totalprice").alias("avg_order_value"),
    max("o_totalprice").alias("max_order")
)
display(df_order_stats)
```

Aggregate with SQL

SQL aggregations work directly in `SELECT` statements:

```
-- Calculate statistics by dealership
SELECT id,
       COUNT(*) AS total_orders,
       SUM(quantity) AS total_quantity,
       AVG(quantity) AS avg_quantity,
       MAX(quantity) AS max_quantity
FROM dealer
GROUP BY id
ORDER BY total_quantity DESC;
```

Use `HAVING` to filter groups after aggregation:

```
-- Find high-volume dealers only
SELECT id, SUM(quantity) AS total_quantity
FROM dealer
GROUP BY id
HAVING SUM(quantity) > 20;
```

Chain transformations for readable pipelines

Spark's lazy evaluation lets you chain multiple transformations into a single, readable pipeline. The operations execute only when you call an action like `display()` or `show()`.

```
from pyspark.sql.functions import col, count

# Chain filter, group, aggregate, and sort
df_result = (
    df_order
    .filter(col("o_orderstatus") == "F")
    .groupBy("o_orderpriority")
    .agg(count("o_orderkey").alias("n_orders"))
```

```
    .sort(col("n_orders").desc())
)
display(df_result)
```

This approach improves readability and lets Spark optimize the entire operation sequence before execution. Each transformation builds on the previous one, creating a clear data flow from source to result.

> **Tip**
>
> When chaining transformations, place filters early in the pipeline to reduce the data volume that subsequent operations must process.

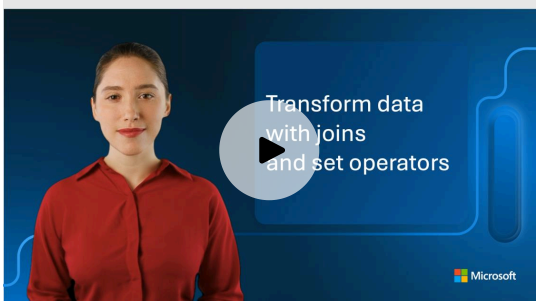
Understanding filtering, grouping, and aggregating prepares you to build efficient transformation pipelines. These operations combine naturally with joins and other transformations to create comprehensive data processing workflows.

6. Transform data with joins and set operators

https://learn.microsoft.com/en-us/training/modules/cleanse-transform-load-data-into-unity-catalog/6-transform-data-join-union-intersect-except

Transform data with joins and set operators

Completed

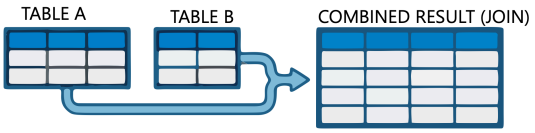


Data engineering workflows often require combining information from multiple datasets. **Joins** combine tables horizontally by matching rows on related columns, while **set operators** combine tables vertically by appending, intersecting, or excluding rows. Understanding when to use each approach helps you build efficient transformation pipelines.

In this unit, you learn how to join tables based on relationships and apply set operators to combine datasets with matching schemas.

Combine tables with joins

Joins merge rows from two tables based on a related column. The join type determines which rows appear in the result and how unmatched rows are handled.



Understand join types

Each join type serves a specific purpose in data transformation:

Join type	Description	Use case
`INNER`	Returns only matching rows from both tables	Link orders with valid customers
`LEFT`	Returns all left table rows; adds NULLs for right table when no match	Include all customers, even without orders
`RIGHT`	Returns all right table rows; adds NULLs for left table when no match	Include all departments, even without employees
`FULL`	Returns all rows from both tables; adds NULLs for non-matches	Combine datasets to find gaps on either side
`SEMI`	Returns left table rows that have matches in the right table	Filter customers who placed orders
`ANTI`	Returns left table rows that have no match in the right table	Find customers who never placed orders
`CROSS`	Returns Cartesian product of all rows	Generate all possible combinations

Join tables with SQL

The `JOIN` clause combines tables based on a condition specified with `ON` or `USING`. Consider employee and department tables where you want to match employees to their departments:

```
-- Inner join: only employees with matching departments
SELECT e.id, e.name, e.deptno, d.deptname
FROM employee e
INNER JOIN department d ON e.deptno = d.deptno;
```

An inner join returns only three rows—employees whose department numbers exist in the department table. Employees in departments 4, 5, and 6 don't appear because those departments aren't in the department table.

When you need to include all employees regardless of department matches, use a left join:

```
-- Left join: all employees, NULL for missing departments
SELECT e.id, e.name, e.deptno, d.deptname
FROM employee e
LEFT JOIN department d ON e.deptno = d.deptno;
```

This returns all six employees. Those without matching departments show NULL in the `deptname` column.

Join DataFrames with PySpark

PySpark's `join()` method provides the same functionality. The `how` parameter specifies the join type:

```
df_employee = spark.table("employee")
df_department = spark.table("department")

# Inner join
df_inner = df_employee.join(
    df_department,
    on=df_employee.deptno == df_department.deptno,
    how="inner"
)
display(df_inner)
```

For left, right, or full outer joins, change the `how` parameter:

```
# Left join: keep all employees
df_left = df_employee.join(
    df_department,
    on="deptno",
    how="left"
)

# Full outer join: keep all rows from both tables
df_full = df_employee.join(
    df_department,
    on="deptno",
    how="full"
)
```

When the join column has the same name in both DataFrames, pass the column name as a string. This approach avoids duplicate columns in the result.

Filter data with semi and anti joins

Semi and anti joins are powerful for filtering data based on existence in another table. A semi join returns rows from the left table that have matches, without including columns from the right table:

```
-- Find employees who belong to listed departments
SELECT *
FROM employee
LEFT SEMI JOIN department ON employee.deptno = department.deptno;
```

An anti join returns the opposite—rows without matches:

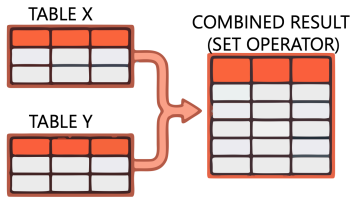
```
-- Find employees in departments not listed
SELECT *
FROM employee
LEFT ANTI JOIN department ON employee.deptno = department.deptno;
```

In PySpark, use `how="semi"` or `how="anti"`:

```
# Employees with matching departments
df_semi = df_employee.join(df_department, on="deptno", how="semi")

# Employees without matching departments
df_anti = df_employee.join(df_department, on="deptno", how="anti")
```

Combine rows with set operators

Set operators work vertically, combining rows from queries with the same column structure. Unlike joins, set operators don't match on conditions—they stack or compare entire result sets.



Understand set operators

Azure Databricks supports three set operators:

Operator	Description	Behavior
`UNION`	Combines rows from both queries	Removes duplicates by default; use `ALL` to keep them
`INTERSECT`	Returns rows that exist in both queries	Only matching rows appear
`EXCEPT`	Returns rows from the first query not in the second	Removes duplicates by default

Both queries must have the same number of columns with compatible data types.

Merge datasets with UNION

`UNION` appends rows from one query to another. By default, it removes duplicates:

```
-- Combine active and archived customers (no duplicates)
SELECT customer_id, name, email FROM active_customers
UNION
SELECT customer_id, name, email FROM archived_customers;
```

To preserve all rows including duplicates, use `UNION ALL`:

```
-- Combine all rows, keeping duplicates
SELECT customer_id, name, email FROM active_customers
UNION ALL
SELECT customer_id, name, email FROM archived_customers;
```

Use `UNION ALL` when you know there are no duplicates or when duplicates are meaningful. It performs better because it skips the deduplication step.

In PySpark, use `union()` for combining DataFrames. Note that PySpark's `union()` behaves like `UNION ALL`—it keeps duplicates:

```
df_active = spark.table("active_customers")
df_archived = spark.table("archived_customers")

# Combine both tables (keeps duplicates)
df_combined = df_active.union(df_archived)

# Remove duplicates if needed
df_distinct = df_active.union(df_archived).distinct()
```

Find common rows with INTERSECT

`INTERSECT` returns only rows that appear in both queries. This helps identify overlapping data:

```
-- Find customers in both active and archived tables
SELECT customer_id FROM active_customers
INTERSECT
SELECT customer_id FROM archived_customers;
```

This query identifies customers who somehow appear in both tables—useful for data quality checks or identifying records that need reconciliation.

Exclude rows with EXCEPT

`EXCEPT` returns rows from the first query that don't exist in the second. This helps identify gaps or unique records:

```
-- Find active customers not in the archived table
SELECT customer_id FROM active_customers
EXCEPT
SELECT customer_id FROM archived_customers;
```

You can also use `MINUS` as a synonym for `EXCEPT`:

```
-- Same result using MINUS syntax
SELECT customer_id FROM active_customers
MINUS
SELECT customer_id FROM archived_customers;
```

Like `UNION`, both `INTERSECT` and `EXCEPT` remove duplicates by default. Add `ALL` to preserve duplicates when needed.

> **Tip**
>
> When chaining set operations, remember that `INTERSECT` has higher precedence than `UNION` and `EXCEPT`. Use parentheses to control evaluation order.

Choose between joins and set operators

The choice between joins and set operators depends on your transformation goal. Joins combine columns from related tables horizontally, while set operators combine rows vertically.

Use **joins** when you need to:

- Enrich records by adding columns from related tables
- Match records based on key relationships
- Filter data based on existence in another table (semi/anti joins)

Use **set operators** when you need to:

- Append records from multiple sources with identical schemas
- Find common records across datasets
- Identify records unique to one dataset

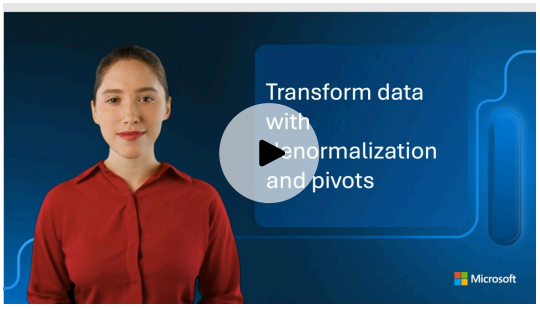
Both approaches form the foundation of effective data transformation pipelines. In the next unit, you explore how to load transformed data into target tables within Unity Catalog.

7. Transform data with denormalization and pivots

https://learn.microsoft.com/en-us/training/modules/cleanse-transform-load-data-into-unity-catalog/7-transform-data-flatten-pivot-unpivot

Transform data with denormalization and pivots

Completed



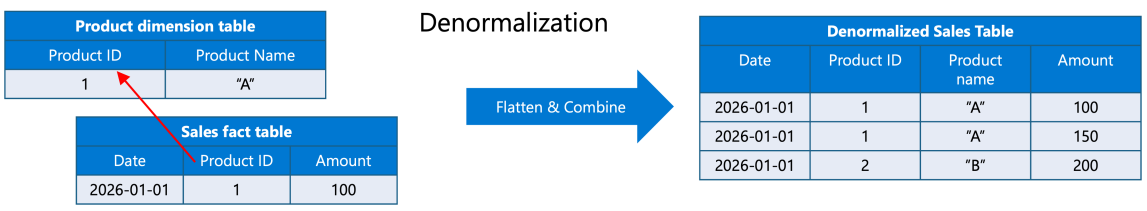
Reshaping data structure is a key part of building effective analytics solutions. **Denormalization**, **pivoting**, and **unpivoting** are transformation techniques that help you reorganize data between normalized and aggregated formats—each serving different analytical and modeling needs.

In this unit, you learn how to flatten normalized tables for faster queries, rotate rows into columns for cross-tabular analysis, and reverse that process when you need to normalize wide datasets.

Understand denormalization

Normalized databases reduce data redundancy by splitting information across multiple related tables. While this design works well for transactional systems, it creates challenges for analytics. Queries must join many tables, which slows performance and increases complexity.

Denormalization addresses these challenges by flattening related tables into fewer, wider tables. You combine data from dimension tables directly into fact tables, eliminating the need for joins at query time.



Consider a sales database with separate tables for orders, customers, and products. A normalized query requires multiple joins:

```
-- Normalized query with multiple joins
SELECT o.order_id,
       o.order_date,
       c.customer_name,
       c.region,
       p.product_name,
       p.category,
       o.quantity,
       o.total_price
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
JOIN products p ON o.product_id = p.product_id;
```

A denormalized table stores this information together:

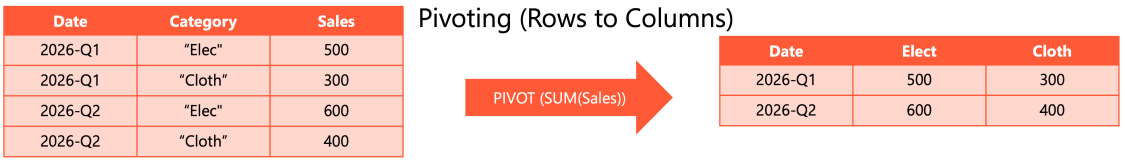
```
-- Create a denormalized orders table
CREATE OR REPLACE TABLE sales.orders_denormalized AS
SELECT o.order_id,
       o.order_date,
       c.customer_name,
       c.region,
       p.product_name,
       p.category,
       o.quantity,
       o.total_price
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
JOIN products p ON o.product_id = p.product_id;
```

Queries against the denormalized table run faster because no joins are required. This approach is common when building star schemas and data marts for business intelligence tools. The tradeoff is increased storage and the need to refresh denormalized tables when source data changes.

> **Tip**
>
> Denormalize data in your gold layer tables when query performance matters more than storage efficiency. Keep normalized structures in bronze and silver layers for flexibility.

Pivot rows into columns

Pivoting transforms unique values from a column into separate columns, converting a "long" table format into a "wide" one. This transformation is particularly useful when preparing data for cross-tabular reports or dashboards.



The `PIVOT` clause requires an aggregate function because multiple source rows might map to the same cell in the pivoted result. You specify which column values become new column names using the `FOR` and `IN` clauses.

Consider quarterly sales data stored in a long format:

```
-- Create sample sales data
CREATE OR REPLACE TEMP VIEW quarterly_sales(year, quarter, region, revenue) AS
VALUES (2024, 1, 'East', 150000),
       (2024, 2, 'East', 175000),
       (2024, 3, 'East', 160000),
       (2024, 4, 'East', 200000),
       (2024, 1, 'West', 180000),
       (2024, 2, 'West', 165000),
       (2024, 3, 'West', 190000),
       (2024, 4, 'West', 210000),
       (2025, 1, 'East', 165000),
       (2025, 2, 'East', 185000);

-- View data in long format
SELECT * FROM quarterly_sales;
```

This query returns revenue as separate rows for each quarter. To compare quarters side by side, pivot the data:

```
-- Pivot quarters into columns
SELECT year, region, Q1, Q2, Q3, Q4
FROM quarterly_sales
PIVOT (
  SUM(revenue)
  FOR quarter IN (1 AS Q1, 2 AS Q2, 3 AS Q3, 4 AS Q4)
);
```

The result shows each region's quarterly revenue as columns, making year-over-year comparisons straightforward:

year	region	Q1	Q2	Q3	Q4
2024	East	150000	175000	160000	200000
2024	West	180000	165000	190000	210000
2025	East	165000	185000	null	null

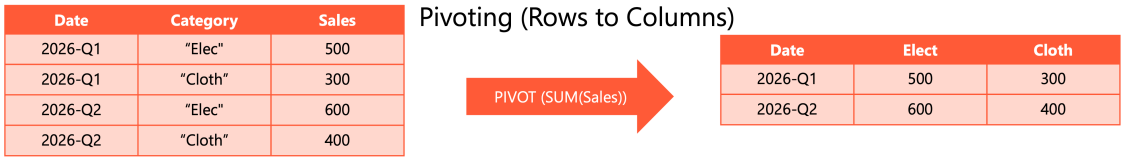
You can apply multiple aggregate functions in a single pivot operation:

```
-- Pivot with multiple aggregations
SELECT year, Q1_total, Q1_avg, Q2_total, Q2_avg
FROM (SELECT year, quarter, revenue FROM quarterly_sales)
PIVOT (
  SUM(revenue) AS total,
  AVG(revenue) AS avg
  FOR quarter IN (1 AS Q1, 2 AS Q2)
);
```

This creates columns for both the sum and average of each quarter's revenue.

Unpivot columns into rows

Unpivoting performs the reverse transformation—it converts columns back into rows. Use unpivoting when you receive wide-format data that needs normalization for analysis or when preparing data for systems that expect a long format.



The `UNPIVOT` clause transforms specified columns into value-name pairs. You define a column to hold the values and another to hold the original column names.

```
-- Create wide-format data
CREATE OR REPLACE TEMP VIEW regional_targets(region, jan, feb, mar, apr) AS
VALUES ('North', 50000, 55000, 52000, 58000),
       ('South', 45000, 48000, 51000, 53000),
       ('East',  60000, 62000, 65000, 68000);

-- Unpivot monthly columns into rows
SELECT *
FROM regional_targets
UNPIVOT (
  target_amount FOR month IN (jan, feb, mar, apr)
);
```

The result converts each month column into rows:

region	month	target_amount
North	jan	50000
North	feb	55000
North	mar	52000
North	apr	58000

region	month	target_amount
South	jan	45000
...	...	...

By default, `UNPIVOT` excludes null values. To include them, specify `INCLUDE NULLS`:

```
-- Unpivot while keeping null values
SELECT *
FROM regional_targets
UNPIVOT INCLUDE NULLS (
  target_amount FOR month IN (jan, feb, mar, apr)
);
```

You can also assign custom aliases to the unpivoted column names:

```
-- Unpivot with custom month labels
SELECT *
FROM regional_targets
UNPIVOT (
  target_amount FOR month IN (
    jan AS `January`,
    feb AS `February`,
    mar AS `March`,
    apr AS `April`
  )
);
```

> **Note**
>
> The `UNPIVOT` clause requires Databricks Runtime 12.2 LTS or later.

Choose the right transformation

Each transformation serves specific analytical needs. Use denormalization when you need fast query performance for dashboards and reports. Choose pivot when analysts need side-by-side comparisons across categories. Apply unpivot when you need to normalize wide data for aggregation or machine learning pipelines.

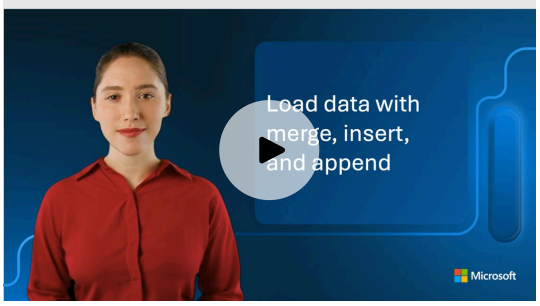
These transformations often work together in data pipelines. You might unpivot source data to standardize its format, apply transformations, and then pivot results for final presentation—or denormalize the output into optimized tables for downstream consumers.

8. Load data with merge, insert, and append

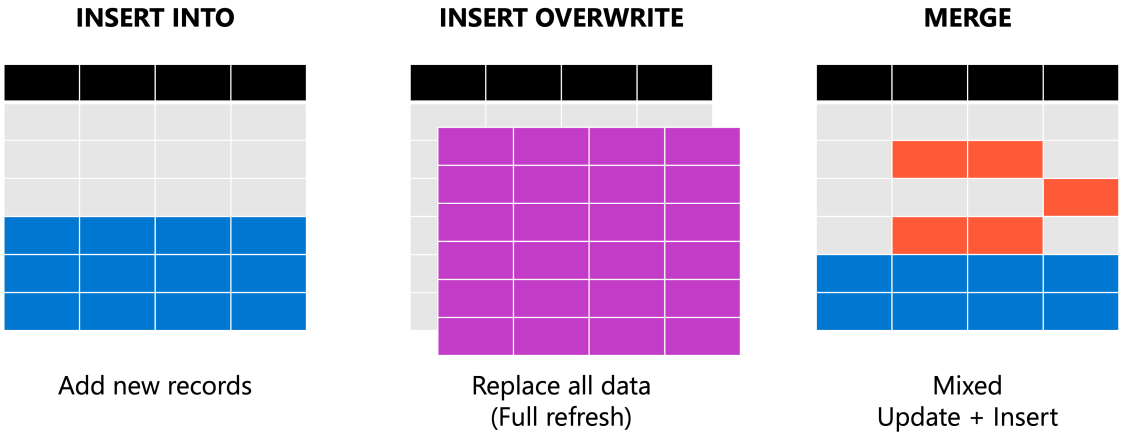
https://learn.microsoft.com/en-us/training/modules/cleanse-transform-load-data-into-unity-catalog/8-load-data-merge-insert-append

Load data with merge, insert, and append

Completed



After you transform and cleanse data, the final step in your pipeline is loading it into target tables. Azure Databricks provides three core loading strategies: **append**, **overwrite**, and **merge**. Each strategy serves different use cases, and choosing the right one depends on how your source data relates to existing records.



In this unit, you learn how to append new rows with INSERT, replace data with overwrite operations, and synchronize changes using MERGE for upsert scenarios.

Append data with INSERT INTO

The most straightforward loading operation adds new rows to an existing table without modifying current data. Use INSERT INTO when you're loading fresh batches of data that don't overlap with existing records—such as daily transaction logs, new event streams, or incremental data extracts.

INSERT INTO works with both SQL and PySpark. The following examples show how to add rows to a sales table.

```
-- Append new records using VALUES
INSERT INTO sales.transactions (transaction_id, amount, transaction_date)
VALUES
    ('TXN001', 150.00, '2026-01-15'),
    ('TXN002', 275.50, '2026-01-15');

-- Append from a staging table or query
INSERT INTO sales.transactions
SELECT * FROM staging.daily_transactions
WHERE transaction_date = current_date();
```

In PySpark, you append data by writing a DataFrame with `append` mode:

```
# Append DataFrame to existing table
df_new_transactions.write.mode("append").saveAsTable("sales.transactions")

# Or use insertInto for existing tables
df_new_transactions.write.insertInto("sales.transactions")
```

> **Tip**
>
> When appending data, ensure your source data is deduplicated before loading. INSERT INTO doesn't check for duplicates—if you load the same batch twice, you get duplicate rows.

Replace data with INSERT OVERWRITE

Sometimes you need to replace existing data rather than add to it. INSERT OVERWRITE truncates the target table or partition before inserting new rows. This approach works well for reprocessing failed batches, refreshing lookup tables, or rebuilding aggregations.

```
-- Replace entire table contents
INSERT OVERWRITE sales.daily_summary
SELECT
    region,
    SUM(amount) as total_sales,
    COUNT(*) as transaction_count
FROM sales.transactions
WHERE transaction_date = current_date()
GROUP BY region;
```

For partitioned tables, you can overwrite specific partitions while leaving others intact:

```
-- Replace only January 2026 partition
INSERT OVERWRITE sales.monthly_report
PARTITION (report_month = '2026-01')
SELECT region, product_category, SUM(amount) as revenue
FROM sales.transactions
```

```
WHERE transaction_date BETWEEN '2026-01-01' AND '2026-01-31'
GROUP BY region, product_category;
```

Delta Lake also supports selective overwrites with the `REPLACE WHERE` clause, which deletes rows matching a condition before inserting:

```
-- Replace transactions for a specific date range
INSERT INTO sales.transactions
REPLACE WHERE transaction_date BETWEEN '2026-01-01' AND '2026-01-07'
SELECT * FROM staging.corrected_transactions;
```

In PySpark, use `overwrite` mode for full table replacement:

```
# Overwrite entire table
df_refreshed.write.mode("overwrite").saveAsTable("sales.daily_summary")

# Overwrite specific partition
df_january.write.mode("overwrite").partitionBy("report_month").saveAsTable("sales.r
```

Important

Overwriting specific partitions by setting `partitionOverwriteMode=dynamic` is a legacy approach that only works on classic compute—it doesn't work on Databricks SQL warehouses or serverless compute. For new pipelines, use `REPLACE USING` instead.

Dynamically replace partitions with REPLACE USING

`REPLACE USING` is the recommended approach for dynamic partition overwrites. It works across all compute types—including Databricks SQL warehouses and serverless compute—and doesn't require Spark session configuration.

```
-- Replace only the partitions touched by the incoming data
INSERT INTO sales.monthly_report
  REPLACE USING (report_month)
  SELECT region, product_category, SUM(amount) AS revenue, report_month
  FROM staging.corrected_transactions
  GROUP BY region, product_category, report_month;
```

The `REPLACE USING` clause atomically deletes rows in `sales.monthly_report` that match incoming rows on `report_month`, then inserts the new rows. Partitions not represented in the source query are left untouched.

When you need more flexible matching—such as replacing rows based on a custom condition or treating NULL values as equal—use `REPLACE ON` instead:

```
-- Replace rows matching a user-defined condition across source and target
INSERT INTO sales.transactions
```

```
  REPLACE ON (target.transaction_id = source.transaction_id)
  SELECT * FROM staging.corrected_transactions AS source;
```

`REPLACE ON` requires Databricks Runtime 17.1 and above. Use `REPLACE WHERE` (covered earlier) when you need condition-based replacement without a source table reference.

Merge data for upsert operations

Real-world data pipelines often need to update existing records while inserting new ones. The MERGE statement handles this pattern—commonly called an "upsert"—by matching source rows to target rows and applying different actions based on whether a match exists.

MERGE is essential for slowly changing dimensions, synchronizing data from source systems, and processing change data capture (CDC) feeds.

```
MERGE INTO customers AS target
USING customer_updates AS source
ON target.customer_id = source.customer_id
WHEN MATCHED THEN
    UPDATE SET
        target.email = source.email,
        target.phone = source.phone,
        target.last_updated = current_timestamp()
WHEN NOT MATCHED THEN
    INSERT (customer_id, name, email, phone, created_date, last_updated)
    VALUES (source.customer_id, source.name, source.email, source.phone,
            current_timestamp(), current_timestamp());
```

This query updates email and phone for existing customers while inserting entirely new customer records. The ON clause defines the matching key—typically a primary key or business identifier.

Merge with conditional logic

Add conditions to WHEN clauses to handle more complex scenarios. For example, you might only update records that have actually changed:

```
MERGE INTO products AS target
USING product_feed AS source
ON target.sku = source.sku
WHEN MATCHED AND source.price <> target.price THEN
    UPDATE SET target.price = source.price, target.updated_at = current_timestamp()
WHEN MATCHED AND source.discontinued = true THEN
    DELETE
WHEN NOT MATCHED THEN
    INSERT *;
```

The `INSERT *` syntax inserts all columns from the source when they match the target schema.

Merge with PySpark

For programmatic control, use the Delta Lake Python API:

```
from delta.tables import DeltaTable

target_table = DeltaTable.forName(spark, "customers")

target_table.alias("target").merge(
    source_df.alias("source"),
    "target.customer_id = source.customer_id"
).whenMatchedUpdate(set={
    "email": "source.email",
    "phone": "source.phone",
    "last_updated": "current_timestamp()"
}).whenNotMatchedInsert(values={
    "customer_id": "source.customer_id",
    "name": "source.name",
    "email": "source.email",
    "phone": "source.phone",
    "created_date": "current_timestamp()",
    "last_updated": "current_timestamp()"
}).execute()
```

For simpler cases where source and target schemas match, use the convenience methods:

```
target_table.alias("target").merge(
    source_df.alias("source"),
    "target.customer_id = source.customer_id"
).whenMatchedUpdateAll(
).whenNotMatchedInsertAll(
).execute()
```

Choose the right loading strategy

Your choice of loading operation depends on the relationship between source and target data:

Scenario	Operation	When to use
New records only	INSERT INTO / append	Daily logs, event streams, incremental extracts
Full refresh	INSERT OVERWRITE	Lookup tables, aggregations, failed batch reprocessing
Mixed updates and inserts	MERGE	CDC feeds, dimension tables, data synchronization
Selective replacement by predicate	REPLACE WHERE	Correcting specific date ranges or logical conditions
Dynamic partition replacement (all compute types)	REPLACE USING	Recommended for partition overwrites on SQL warehouses and serverless

Scenario	Operation	When to use
Custom-condition row replacement	REPLACE ON	Partition overwrites with NULL-equality or cross-column conditions

Important

MERGE operations require a unique match between source and target rows. If multiple source rows match the same target row, the operation fails. Deduplicate your source data before merging.

Understanding these loading patterns prepares you to build reliable data pipelines that maintain data integrity while efficiently processing both new and changed records.

9. Exercise - Cleanse, Transform, and Load Data into Unity Catalog

https://learn.microsoft.com/en-us/training/modules/cleanse-transform-load-data-into-unity-catalog/9-exercise-cleanse-transform-load-data-into-unity-catalog

Exercise - Cleanse, Transform, and Load Data into Unity Catalog

Completed

Now it's your chance to cleanse, transform, and load data into Unity Catalog. In this lab, you clean and reshape raw real estate data in Azure Databricks. You choose the right data types for prices and timestamps, remove duplicate listings and fill missing values using PySpark, and combine data across tables with inner and left joins. You also use SQL PIVOT and UNPIVOT to restructure market statistics for trend analysis.

Note

To complete this lab, you need an Azure subscription in which you have administrative access.

Launch the exercise and follow the instructions.

Launch Exercise

10. Module assessment

https://learn.microsoft.com/en-us/training/modules/cleanse-transform-load-data-into-unity-catalog/10-knowledge-check

Module assessment

Completed

11. Summary

https://learn.microsoft.com/en-us/training/modules/cleanse-transform-load-data-into-unity-catalog/11-summary

Summary

Completed

Throughout this module, you explored the complete data engineering workflow for cleansing, transforming, and loading data into **Unity Catalog** tables in Azure Databricks. From initial data profiling to final loading operations, each technique builds on the previous to deliver high-quality, well-structured data ready for analysis.

You learned how **data profiling** using `ANALYZE TABLE` and Unity Catalog's monitoring features reveals data quality issues like null percentages, duplicate records, and distribution anomalies. You explored how selecting **appropriate data types** affects storage efficiency, query performance, and data integrity—particularly the importance of using `DECIMAL` for financial calculations. You practiced identifying and resolving **duplicates and null values** using techniques like `QUALIFY` with window functions, `dropDuplicates()`, and `fillna()` methods.

For data transformation, you applied **filtering, grouping, and aggregation** operations to shape data for analytical requirements. You combined datasets using **joins** for horizontal merging and **set operators** like `UNION`, `INTERSECT`, and `EXCEPT` for vertical combinations. You reshaped data structures through **denormalization** for query performance, **pivoting** for cross-tabular analysis, and **unpivoting** to normalize wide datasets.

Finally, you implemented **loading strategies** that match your data scenarios: `INSERT INTO` for appending new records, `INSERT OVERWRITE` for replacing data, and `MERGE INTO` for upsert operations that synchronize changes. As you build data pipelines, apply these techniques

systematically—profile first to understand your data, cleanse to resolve quality issues, transform to meet analytical requirements, and load with the appropriate strategy to maintain data integrity in your lakehouse.

Implement and manage data quality constraints with Azure Databricks

1. Introduction

https://learn.microsoft.com/en-us/training/modules/implement-manage-data-quality-constraints-unity-catalog/1-introduction

Introduction

Completed

Data quality issues can derail analytics projects, corrupt business reports, and erode confidence in your data platform. When invalid data enters your tables—whether through type mismatches, missing required values, or unexpected schema changes—problems compound as that data flows through pipelines and reaches downstream consumers. Implementing robust **data quality constraints** at the point of ingestion creates a foundation of trust in your data assets.

Azure Databricks provides multiple mechanisms for enforcing data quality within **Unity Catalog**. **Schema enforcement** validates data types automatically when writing to Delta Lake tables. **Table constraints** define rules that reject invalid records at write time. **Pipeline expectations** in Lakeflow Spark Declarative Pipelines enable real-time quality checks on streaming data with configurable actions for violations. Together, these capabilities form a comprehensive approach to maintaining data integrity.

Throughout this module, you explore practical techniques for implementing data quality constraints. You learn how to enforce data type checks using **schema validation**, **explicit casting**, and **CHECK constraints**. You discover strategies for managing **schema drift** when source systems evolve over time. You implement validation checks for **nullability**, **uniqueness**, and **value ranges**. Finally, you

master pipeline expectations that monitor data quality metrics and take automated actions when violations occur.

By combining these approaches, you build pipelines that catch quality issues early, prevent invalid data from reaching production tables, and provide visibility into the health of your data. These skills are essential for data engineers working with Unity Catalog who need to maintain high data quality standards across their organization's data estate.

2. Implement validation checks

https://learn.microsoft.com/en-us/training/modules/implement-manage-data-quality-constraints-unity-catalog/2-implement-validation-checks

Implement validation checks

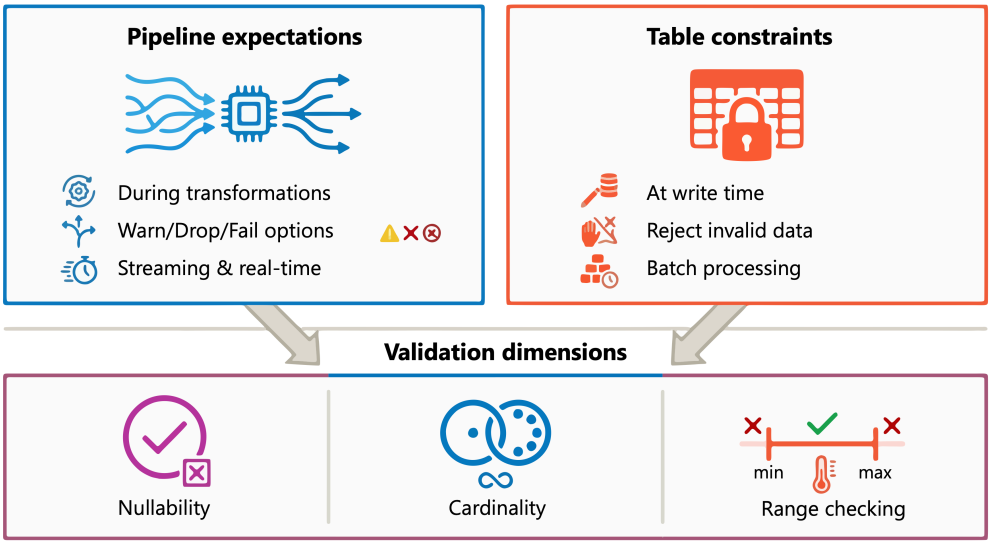
Completed

Data pipelines often encounter records with missing values, duplicate identifiers, or values that fall outside acceptable ranges. Without validation checks in place, these data quality issues propagate downstream, causing incorrect analytics, failed reports, and unreliable business decisions. Validation checks catch these problems at the point of data ingestion, ensuring that only quality data flows through your pipeline.

In this unit, you learn how to implement validation checks for nullability, data cardinality, and range checking using Lakeflow Spark Declarative Pipelines expectations and Delta Lake table constraints.

Understand validation approaches in Azure Databricks

Azure Databricks provides two primary mechanisms for implementing validation checks: pipeline expectations and table constraints. Each approach serves different scenarios and offers distinct capabilities.



Pipeline expectations apply validation during data transformations in Lakeflow Spark Declarative Pipelines. Expectations let you warn, drop invalid records, or fail the pipeline when data violates your rules. This approach works well for streaming tables and materialized views where you need real-time quality control.

Table constraints enforce rules directly on Delta Lake tables. Constraints reject invalid data at write time, preventing bad records from ever entering your tables. This approach suits batch processing and scenarios requiring strict data integrity guarantees.

With these approaches, you can validate three key dimensions of data quality: nullability (ensuring required values exist), data cardinality (verifying uniqueness where expected), and range checking (confirming values fall within acceptable bounds).

Implement nullability checks

Nullability validation ensures that required columns contain values. Consider a customer table where `email` and `customer_id` must always have values. You can implement these checks using expectations in your pipeline.

Using Python, add expectation decorators to your streaming table definition:

```
from pyspark import pipelines as dp

@dp.table()
@dp.expect_or_drop("valid_email", "email IS NOT NULL")
@dp.expect_or_drop("valid_customer_id", "customer_id IS NOT NULL")
def customers():
    return spark.readStream.table("raw.customers")
```

The same validation in SQL uses the `CONSTRAINT` clause:

```
CREATE OR REFRESH STREAMING TABLE customers(
    CONSTRAINT valid_email EXPECT (email IS NOT NULL) ON VIOLATION DROP ROW,
    CONSTRAINT valid_customer_id EXPECT (customer_id IS NOT NULL) ON VIOLATION DROP
) AS SELECT * FROM STREAM(raw.customers);
```

For Delta Lake tables outside of pipelines, use `NOT NULL` constraints in the table definition:

```
CREATE TABLE customers (
    customer_id INT NOT NULL,
    email STRING NOT NULL,
    first_name STRING,
    last_name STRING
);
```

You can also add nullability constraints to existing tables:

```
ALTER TABLE customers ALTER COLUMN email SET NOT NULL;
```

> **Note**
>
> Before adding a `NOT NULL` constraint to an existing table, Azure Databricks verifies that all existing rows satisfy the constraint. The operation fails if any null values exist in the column.

Validate data cardinality

Cardinality validation ensures that columns expected to contain unique values actually do. This check is essential for primary keys, transaction identifiers, and other fields where duplicates indicate data quality problems.

Pipeline expectations can validate cardinality by checking for conditions that indicate uniqueness issues. For example, you can verify that a Social Security Number appears only once per person:

```
from pyspark.sql.window import Window
from pyspark.sql.functions import count

@dp.table()
@dp.expect("unique_ssn_per_person", "ssn_count = 1")
def employees():
    df = spark.table("raw.employees")
    w = Window.partitionBy("ssn")
    return df.withColumn("ssn_count", count("*").over(w))
```

For more comprehensive cardinality checks, combine expectations with aggregation logic in your transformation:

```
CREATE OR REFRESH MATERIALIZED VIEW order_validation AS
SELECT
    order_id,
    COUNT(*) as occurrence_count
FROM orders
GROUP BY order_id
HAVING COUNT(*) > 1;
```

This materialized view identifies any duplicate order IDs, enabling you to investigate and resolve cardinality issues.

Delta Lake supports primary key constraints that document expected uniqueness, though these constraints are informational and not enforced:

```
CREATE TABLE orders (
    order_id INT NOT NULL,
    customer_id INT,
    order_date DATE,
    CONSTRAINT orders_pk PRIMARY KEY (order_id)
);
```

> **Tip**
>
> While primary key constraints aren't enforced, they help query optimization and document your data model's intended structure. Use pipeline expectations to actively enforce uniqueness.

Apply range checking

Range validation confirms that numeric, date, and other values fall within acceptable bounds. This check catches data entry errors, system glitches, and integration issues that produce out-of-range values.

Define range expectations using comparison operators or the `BETWEEN` clause:

```
@dp.table()
@dp.expect_or_fail("valid_age", "age BETWEEN 0 AND 150")
@dp.expect_or_fail("valid_salary", "salary >= 0")
@dp.expect_or_fail("valid_hire_date", "hire_date <= current_date()")
def employees():
    return spark.readStream.table("raw.employees")
```

In SQL, apply the same range checks:

```
CREATE OR REFRESH STREAMING TABLE transactions(
    CONSTRAINT valid_amount EXPECT (amount > 0) ON VIOLATION DROP ROW,
    CONSTRAINT valid_date EXPECT (transaction_date <= current_date()) ON VIOLATION
```

```
    CONSTRAINT valid_quantity EXPECT (quantity BETWEEN 1 AND 10000) ON VIOLATION DR
) AS SELECT * FROM STREAM(raw.transactions);
```

For Delta Lake tables, use `CHECK` constraints to enforce ranges at write time:

```
ALTER TABLE employees ADD CONSTRAINT valid_age CHECK (age >= 18 AND age <= 120);
ALTER TABLE transactions ADD CONSTRAINT positive_amount CHECK (amount > 0);
```

Range checks can also validate business rules that combine multiple conditions:

```
@dp.expect("valid_discount", """
    discount_percent >= 0
    AND discount_percent <= 100
    AND (discount_percent <= 50 OR customer_tier = 'PREMIUM')
""")
```

This expectation ensures discounts stay within bounds and enforces that only premium customers receive discounts over 50%.

Choose actions for validation failures

When validation fails, you choose how your pipeline responds. Each action suits different business requirements and data criticality levels.

Action	Use case	Behavior
Warn (default)	Monitoring and analysis	Invalid records write to target; metrics logged
Drop	Data cleansing	Invalid records removed before write
Fail	Critical data integrity	Pipeline stops; transaction rolls back

Use the **warn** action when you need visibility into data quality issues without blocking data flow:

```
@dp.expect("has_phone", "phone_number IS NOT NULL")
```

Use **drop** when invalid records should be filtered out silently:

```
@dp.expect_or_drop("complete_address", "street IS NOT NULL AND city IS NOT NULL")
```

Use **fail** when data integrity is critical and invalid records are unacceptable:

```
@dp.expect_or_fail("valid_account_balance", "balance >= 0")
```

You can view expectation metrics in the pipeline UI by selecting a dataset with expectations and opening the **Data quality** tab. These metrics help you monitor validation pass rates and identify systematic data quality issues.

Now that you understand how to implement validation checks, you can ensure your data pipelines maintain quality standards for nullability, cardinality, and value ranges.

3. Implement data type checks

https://learn.microsoft.com/en-us/training/modules/implement-manage-data-quality-constraints-unity-catalog/3-implement-data-type-checks

Implement data type checks

Completed

Every column in your dataset has an expected data type, and when incoming data doesn't match that expectation, problems cascade downstream. A string value where an integer belongs can break calculations, corrupt aggregations, or cause pipeline failures. Data type checks ensure that each field contains values of the correct type before they enter your tables.

In this unit, you learn how to implement data type checks using schema enforcement, explicit type casting, and validation constraints in Azure Databricks.

Understand schema enforcement

Delta Lake validates data types automatically when you write data to a table. This built-in mechanism, called **schema enforcement**, compares the data types of incoming columns against the target table's schema. When a type mismatch occurs, Delta Lake attempts to safely cast the value. If the cast fails, the write operation raises an error.

Schema enforcement applies the following rules during insert operations:

- All inserted columns must exist in the target table
- All column data types must match or be safely castable to the target types

Consider a table with an integer column:

```
CREATE TABLE inventory (
    product_id INT,
    quantity INT,
    last_updated DATE
);
```

When you insert data where `quantity` is a string that represents a number, Delta Lake casts it to the target type:

```
-- This succeeds because '100' can be cast to INT
INSERT INTO inventory VALUES (1, '100', '2026-01-15');
```

However, inserting a string that can't be converted to an integer causes the operation to fail:

```
-- This fails because 'fifty' cannot be cast to INT
INSERT INTO inventory VALUES (2, 'fifty', '2026-01-15');
```

Schema enforcement provides a first line of defense against type mismatches. For more control over how mismatches are handled, you can use explicit casting.

Use explicit type casting

The `cast` function converts values from one data type to another. When the conversion fails, an error is raised. The `try_cast` function works the same way but returns `NULL` instead of raising an error when the conversion fails.

Use `cast` when you want strict validation that stops processing on invalid data:

```
SELECT
    cast(raw_amount AS DECIMAL(10,2)) AS amount,
    cast(raw_date AS DATE) AS transaction_date
FROM staging_data;
```

Use `try_cast` when you want to identify invalid values without failing the query:

```
SELECT
    raw_amount,
    try_cast(raw_amount AS DECIMAL(10,2)) AS validated_amount,
    CASE
        WHEN try_cast(raw_amount AS DECIMAL(10,2)) IS NULL
        THEN 'Invalid amount format'
        ELSE 'Valid'
    END AS validation_status
FROM staging_data;
```

With `try_cast`, records with invalid values return `NULL` for the cast column. You can then filter, flag, or quarantine these records based on your data quality requirements.

Implement type validation with constraints

CHECK constraints let you define custom validation rules that are enforced whenever data is inserted or updated. While typically used for value ranges and patterns, you can combine them with type-aware functions to create sophisticated type checks.

For example, you can validate that a string column contains only numeric characters:

```
CREATE TABLE orders (
    order_id INT,
    order_total STRING
);

ALTER TABLE orders
ADD CONSTRAINT valid_order_total CHECK (order_total REGEXP '^[0-9]+(\\.[0-9]+)?$')
```

This constraint ensures the `order_total` column contains values that look like valid numbers, catching malformed data before it enters the table.

For date validation, you can use the `try_cast` function within a constraint:

```
ALTER TABLE events
ADD CONSTRAINT valid_event_date CHECK (try_cast(event_date_str AS DATE) IS NOT NULL
```

This approach rejects records where `event_date_str` doesn't contain a valid date format.

> **Important**
>
> Before adding a constraint to an existing table, Azure Databricks verifies that all existing rows satisfy the constraint. Plan for this validation step, especially on large tables.

Handle type mismatches in pipelines

When processing data from external sources, type mismatches are common. Implement a pattern that separates valid from invalid records, allowing you to process good data while quarantining problematic records for review:

```
-- Insert valid records into the target table
INSERT INTO silver_transactions
SELECT
    transaction_id,
    cast(amount AS DECIMAL(10,2)) AS amount,
    cast(transaction_date AS DATE) AS transaction_date
FROM bronze_transactions
WHERE try_cast(amount AS DECIMAL(10,2)) IS NOT NULL
  AND try_cast(transaction_date AS DATE) IS NOT NULL;
```

```
-- Capture invalid records for investigation
INSERT INTO quarantine_transactions
SELECT
    transaction_id,
    amount AS raw_amount,
    transaction_date AS raw_date,
    current_timestamp() AS quarantined_at,
    'Type validation failed' AS reason
FROM bronze_transactions
WHERE try_cast(amount AS DECIMAL(10,2)) IS NULL
   OR try_cast(transaction_date AS DATE) IS NULL;
```

This pattern ensures your pipeline continues processing valid data while preserving invalid records for later analysis.

Use pipeline expectations for type checking

Lakeflow Spark Declarative Pipelines provides **expectations** that allow you to define data quality rules directly in your pipeline definitions. You can use expectations to check that values can be cast to expected types:

```
from pyspark import pipelines as dp

@dp.table
@dp.expect_or_drop("valid_amount", "try_cast(amount AS DECIMAL(10,2)) IS NOT NULL")
@dp.expect_or_drop("valid_date", "try_cast(event_date AS DATE) IS NOT NULL")
def validated_transactions():
    return spark.readStream.table("raw_transactions")
```

With `expect_or_drop`, records that fail the type check are dropped before reaching the target table. Use `expect` to log violations without dropping records, or `expect_or_fail` to stop the pipeline when violations occur.

For SQL-based pipelines:

```
CREATE OR REFRESH STREAMING TABLE validated_transactions (
    CONSTRAINT valid_amount EXPECT (try_cast(amount AS DECIMAL(10,2)) IS NOT NULL)
    CONSTRAINT valid_date EXPECT (try_cast(event_date AS DATE) IS NOT NULL) ON VIOL
)
AS SELECT * FROM STREAM(raw_transactions);
```

Pipeline expectations integrate data type validation directly into your ETL logic, providing visibility into data quality metrics through the pipeline UI.

Implementing data type checks at multiple levels—schema enforcement, explicit casting, constraints, and pipeline expectations—creates defense in depth. Each layer catches issues that might slip past others, resulting in higher data quality in your Unity Catalog tables.

4. Detect and manage schema drift

https://learn.microsoft.com/en-us/training/modules/implement-manage-data-quality-constraints-unity-catalog/4-detect-manage-schema-drift

Detect and manage schema drift

Completed

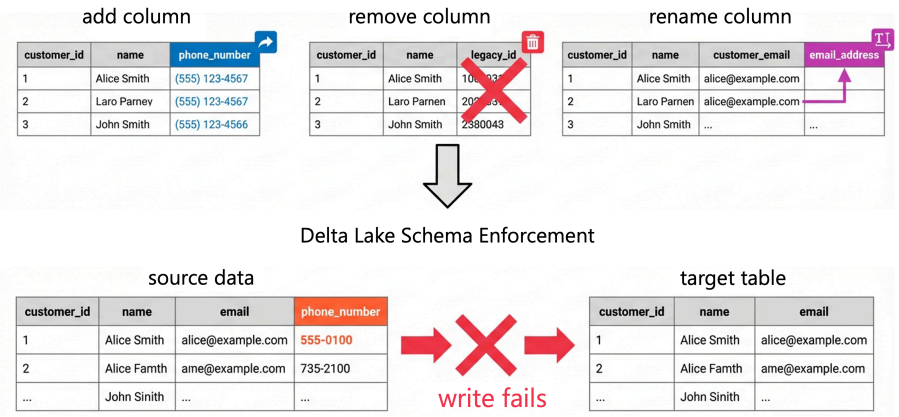
Data pipelines often receive data from sources that evolve over time. New columns appear, others disappear, and the structure of incoming data changes as business requirements shift. Without proper controls, these changes can silently corrupt your data or break your pipelines entirely.

In this unit, you learn how to detect and manage schema drift—the structural changes that occur when source systems add, remove, or rename columns over time.

Recognize schema drift challenges

While data type validation ensures values match expected types (as covered in the previous unit), schema drift addresses a different challenge: the structure of your data changes over time. A source system adds a new `phone_number` column, removes a deprecated `legacy_id` field, or renames `customer_email` to `email_address`. These structural changes happen independently of type validation.

Delta Lake's schema enforcement blocks structural mismatches by default. When incoming data contains columns not present in the target table, or when required columns are missing, the write operation fails. This fail-fast behavior protects your tables from unexpected structural changes, but you need strategies to handle legitimate schema evolution.



Consider a streaming pipeline that processes customer data:

```
CREATE TABLE customers (
    customer_id INT,
    name STRING,
    email STRING
);
```

When your source system adds a `phone_number` column and starts sending it in the data feed, writes fail because the target table doesn't include this column. Your pipeline stops until you decide how to handle the new field—either by rejecting it, adding it to the table schema, or preserving it for later analysis.

Detect and respond to schema drift

Schema drift occurs when the structure of incoming data changes over time. Your data pipelines must detect these changes and respond appropriately—either by adapting to the new schema or by alerting operators to investigate.

Azure Databricks provides several approaches for handling schema drift depending on your requirements.

Fail on schema mismatch

The strictest approach stops processing when schema drift is detected. This strategy works well when data structure changes require manual review before pipeline updates.

With Auto Loader, you can configure this behavior using the `failOnNewColumns` mode:

```
(spark.readStream
    .format("cloudFiles")
    .option("cloudFiles.format", "json")
    .option("cloudFiles.schemaLocation", "/path/to/schema")
    .option("cloudFiles.schemaEvolutionMode", "failOnNewColumns")
```

```
    .load("/path/to/source")
    .writeStream
    .option("checkpointLocation", "/path/to/checkpoint")
    .toTable("target_table")
)
```

When Auto Loader encounters a new column not defined in the schema, the stream stops with an `UnknownFieldException`. You must update the schema or remove the offending data before the pipeline can resume.

Automatically adapt to new columns

For scenarios where adding new columns is acceptable, you can enable schema evolution. This approach automatically adds new columns to the target table without manual intervention.

Enable schema evolution for write operations:

```
(df.write
    .format("delta")
    .mode("append")
    .option("mergeSchema", "true")
    .saveAsTable("target_table")
)
```

For `MERGE` operations, use the schema evolution syntax:

```
MERGE WITH SCHEMA EVOLUTION INTO target_table t
USING source_data s
ON t.customer_id = s.customer_id
WHEN MATCHED THEN UPDATE SET *
WHEN NOT MATCHED THEN INSERT *
```

New columns from the source are automatically added to the target table. However, you should carefully consider when to use schema evolution—automatic changes might introduce columns you didn't expect.

Rescue unexpected data

When you need to preserve data that doesn't match your schema without modifying the target structure, use the rescued data column. This approach captures mismatched data in a separate JSON column for later review.

```
(spark.readStream
    .format("cloudFiles")
    .option("cloudFiles.format", "json")
    .option("cloudFiles.schemaEvolutionMode", "rescue")
    .option("rescuedDataColumn", "_rescued_data")
    .load("/path/to/source")
    .writeStream
```

```
    .option("checkpointLocation", "/path/to/checkpoint")
    .toTable("target_table")
)
```

Data that doesn't match the schema—including new columns, type mismatches, and case differences—is stored in the `_rescued_data` column. Your pipeline continues processing without interruption, and you can analyze rescued data to understand schema changes.

Implement error handling strategies

Beyond automatic schema management, you need strategies for handling records that violate your data quality rules. Lakeflow Spark Declarative Pipelines provides expectations that let you define constraints and specify how to handle violations.

Drop invalid records

Use expectations to filter out records that don't meet your structure requirements:

```
import databricks.sdk.pipelines as dp

@dp.table
@dp.expect_or_drop("required_columns_present",
    "customer_id IS NOT NULL AND email IS NOT NULL")
def validated_customers():
    return spark.readStream.table("raw_customers")
```

Records missing required columns are dropped before reaching the target table. The pipeline tracks metrics showing how many records were dropped.

Fail on critical violations

When data structure issues indicate a serious problem, configure expectations to halt processing:

```
CREATE OR REFRESH STREAMING TABLE validated_orders (
    CONSTRAINT valid_structure
    EXPECT (order_id IS NOT NULL AND product_id IS NOT NULL)
    ON VIOLATION FAIL UPDATE
) AS SELECT * FROM STREAM(raw_orders);
```

The pipeline stops immediately when it encounters records missing required columns, preventing invalid data from propagating downstream.

Quarantine problematic records

For a balanced approach, route invalid records to a separate quarantine table while allowing valid records to proceed:

```
import databricks.sdk.pipelines as dp

@dp.table
def valid_transactions():
    return (
        spark.readStream.table("raw_transactions")
        .filter("amount IS NOT NULL AND transaction_date IS NOT NULL")
    )

@dp.table
def quarantine_transactions():
    return (
        spark.readStream.table("raw_transactions")
        .filter("amount IS NULL OR transaction_date IS NULL")
    )
```

This pattern lets you investigate problematic records without blocking your main data flow.

Apply best practices for schema management

Managing schema effectively requires a combination of technical controls and operational practices.

Document your expected schemas explicitly. Store schema definitions in version control alongside your pipeline code. When source systems change, you have a clear reference for understanding the impact.

Implement alerting for schema drift. Configure your pipelines to send notifications when schema changes are detected, even if the pipeline handles them automatically. Early awareness helps you understand evolving data patterns.

Use consistent naming conventions. Standardize column names across your data estate to reduce case-sensitivity issues and make schema validation more predictable.

Test schema handling in development. Before deploying pipelines to production, verify that your schema enforcement and drift handling work as expected with test data that includes edge cases.

Consider the tradeoffs between strictness and flexibility. Strict schema enforcement catches problems early but requires more maintenance. Flexible approaches like schema evolution reduce operational overhead but might allow unexpected changes.

With these strategies in place, your data pipelines can maintain data quality while adapting to the inevitable changes in your source systems.

5. Manage data quality with pipeline expectations

https://learn.microsoft.com/en-us/training/modules/implement-manage-data-quality-constraints-unity-catalog/5-manage-data-quality-pipeline-expectations

Manage data quality with pipeline expectations

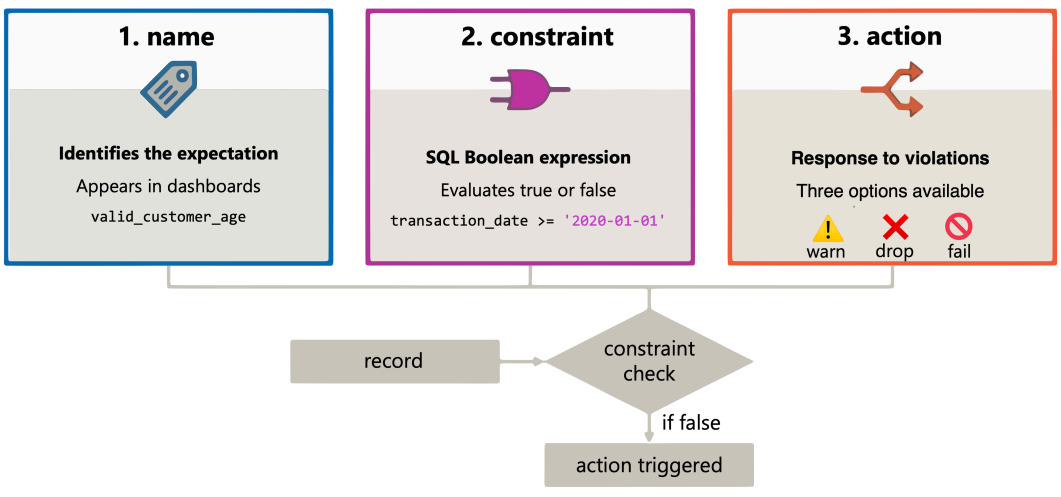
Completed

Data quality issues can disrupt downstream analytics and erode trust in your data. When invalid records slip into production tables, they might cause report failures, incorrect business decisions, or compliance violations. Pipeline expectations in Lakeflow Spark Declarative Pipelines give you a way to define quality rules that run automatically as data flows through your pipeline.

With expectations, you specify what valid data looks like using SQL constraints. The pipeline then checks every record against these rules and takes the action you configure—logging the issue, dropping the record, or failing the update entirely.

Define expectations with three components

Every expectation consists of three parts: a name, a constraint, and an action. Understanding these components helps you design effective data quality checks.



The **name** identifies the expectation and appears in monitoring dashboards. Choose names that clearly describe what you're validating. For example, `valid_customer_age` communicates the rule's purpose better than `check_1`.

The **constraint** is a SQL Boolean expression that evaluates to true or false for each record. When a record fails the constraint, the expectation triggers. You can use any valid SQL syntax except custom Python functions, external service calls, or subqueries.

Here's how you might check that transaction dates fall within an expected range:

```
@dp.expect("valid_transaction_date", "transaction_date >= '2020-01-01' AND transact
```

The same constraint in SQL:

```
CONSTRAINT valid_transaction_date EXPECT (transaction_date >= '2020-01-01' AND tra
```

The **action** determines what happens when a record violates the constraint. You have three options, each suited to different scenarios in your data pipeline.

Choose the right action for invalid records

Your choice of action depends on how critical the data quality issue is and how you want to handle violations.

Action	When to use	Behavior
Warn	Monitoring quality trends without blocking data	Invalid records flow through; metrics captured for review
Drop	Filtering out bad records automatically	Invalid records removed; valid records continue
Fail	Stopping the pipeline for critical violations	Update fails immediately; requires manual intervention

The **warn** action is the default behavior. It writes all records to the target table, including invalid ones, while logging metrics about failures. This approach works well when you're establishing baselines or when downstream systems can handle some data quality issues.

```
@dp.expect("non_null_email", "email IS NOT NULL")
def customer_contacts():
    return spark.readStream.table("raw.customers")
```

The **drop** action removes records that fail validation before writing to the target. Use this when you need clean data downstream and can afford to lose some records.

```
@dp.expect_or_drop("valid_price", "price >= 0 AND price <= 10000")
def validated_orders():
```

```
    return spark.readStream.table("raw.orders")
```

The SQL equivalent uses `ON VIOLATION DROP ROW`:

```
CREATE OR REFRESH STREAMING TABLE validated_orders(
    CONSTRAINT valid_price EXPECT (price >= 0 AND price <= 10000) ON VIOLATION DROP
) AS SELECT * FROM STREAM(raw.orders);
```

The **fail** action stops execution immediately when any record violates the constraint. The pipeline atomically rolls back any partial updates. This option makes sense for critical data where invalid records indicate a serious upstream problem.

```
@dp.expect_or_fail("required_customer_id", "customer_id IS NOT NULL")
def critical_transactions():
    return spark.readStream.table("raw.transactions")
```

In SQL, use `ON VIOLATION FAIL UPDATE`:

```
CREATE OR REFRESH STREAMING TABLE critical_transactions(
    CONSTRAINT required_customer_id EXPECT (customer_id IS NOT NULL) ON VIOLATION F
) AS SELECT * FROM STREAM(raw.transactions);
```

> **Note**
>
> When a pipeline has multiple parallel flows, a failure in one flow doesn't cause other flows to fail. Each flow operates independently.

Combine multiple expectations for comprehensive checks

Real-world data validation often requires multiple rules. You can stack expectations on a single dataset to check different aspects of data quality.

In Python, add multiple decorator calls:

```
@dp.expect("valid_amount", "amount > 0")
@dp.expect("valid_currency", "currency IN ('USD', 'EUR', 'GBP')")
@dp.expect("valid_timestamp", "created_at <= current_timestamp()")
def validated_payments():
    return spark.readStream.table("raw.payments")
```

Python also supports grouping expectations into reusable dictionaries. This approach helps you maintain consistency across multiple tables:

```
payment_rules = {
    "valid_amount": "amount > 0",
    "valid_currency": "currency IN ('USD', 'EUR', 'GBP')",
```

```
    "valid_timestamp": "created_at <= current_timestamp()"
}

@dp.expect_all_or_drop(payment_rules)
def clean_payments():
    return spark.readStream.table("raw.payments")
```

The `expect_all`, `expect_all_or_drop`, and `expect_all_or_fail` decorators apply the same action to all expectations in the group.

In SQL, separate multiple constraints with commas:

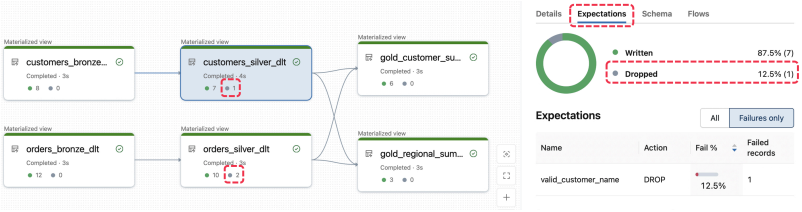
```
CREATE OR REFRESH STREAMING TABLE validated_payments(
    CONSTRAINT valid_amount EXPECT (amount > 0),
    CONSTRAINT valid_currency EXPECT (currency IN ('USD', 'EUR', 'GBP')),
    CONSTRAINT valid_timestamp EXPECT (created_at <= current_timestamp())
) AS SELECT * FROM STREAM(raw.payments);
```

Monitor expectation results

After you define expectations, you can track their results through the pipeline UI. This visibility helps you understand data quality trends and identify recurring issues.

To view expectation metrics:

1. Open **Jobs & Pipelines** in the Azure Databricks workspace sidebar.
2. Select your pipeline by name.
3. Select a dataset that has expectations defined.
4. Open the **Data quality** tab in the right sidebar.



The metrics show you how many records passed or failed each expectation during pipeline runs. For `warn` and `drop` actions, you see counts of violations. For `fail` actions, the pipeline stops before metrics are recorded, but error messages include details about the violating record.

When a `fail` expectation triggers, the error message provides context to help you investigate:

```
[EXPECTATION_VIOLATION.VERBOSITY_ALL] Flow 'sensor-pipeline' failed to meet
the expectation. Violated expectations: 'temperature_in_valid_range'.
Input data: '{"id":"TEMP_001","temperature":-500,"timestamp_ms":"1710498600"}'.
```

You can also query expectation metrics programmatically through the Lakeflow Spark Declarative Pipelines event log. This approach enables you to build custom dashboards or trigger alerts based on

data quality thresholds.

Understanding when to use expectations sets you up to build pipelines that catch data quality issues before they affect downstream consumers. With the right combination of constraints and actions, you create a reliable data foundation for your analytics workloads.

6. Exercise - Implement and Manage Data Quality Constraints with Azure Databricks

https://learn.microsoft.com/en-us/training/modules/implement-manage-data-quality-constraints-unity-catalog/6-exercise-implement-manage-data-quality-constraints-unity-catalog

Exercise - Implement and Manage Data Quality Constraints with Azure Databricks

Completed

Now it's your chance to implement and manage data quality constraints with Azure Databricks. In this lab, you build a Lakeflow Spark Declarative Pipeline for ClearCover Insurance that enforces data quality constraints on raw claims data. You implement nullability and range checks using pipeline expectations, validate data types with col().cast(), and handle schema drift using Auto Loader's rescued data column. You then create and run the pipeline in the Databricks UI and monitor data quality metrics.

> **Note**
>
> To complete this lab, you need an Azure subscription in which you have administrative access.

Launch the exercise and follow the instructions.

Launch Exercise

7. Module assessment

https://learn.microsoft.com/en-us/training/modules/implement-manage-data-quality-constraints-unity-catalog/7-knowledge-check

Module assessment

Completed

8. Summary

https://learn.microsoft.com/en-us/training/modules/implement-manage-data-quality-constraints-unity-catalog/8-summary

Summary

Completed

Maintaining data quality requires a multi-layered approach that catches issues at every stage of your data pipeline. Throughout this module, you explored the tools Azure Databricks provides for implementing robust data quality constraints in Unity Catalog—from **schema enforcement** that validates data types automatically to **pipeline expectations** that monitor quality in real time.

You learned how Delta Lake's built-in schema enforcement acts as a first line of defense, rejecting writes when data types cannot be safely cast. For more nuanced type handling, **explicit casting** with `cast()` and `try_cast()` gives you control over how mismatches are resolved. **CHECK constraints** enforce business rules directly on your tables, ensuring invalid data never enters production.

Schema drift is inevitable when working with evolving data sources. You explored strategies for handling this reality—failing fast when changes require review, enabling **schema evolution** to adapt automatically, or using **rescued data columns** to preserve unexpected data for investigation. These options let you balance strictness with flexibility based on your pipeline's requirements.

Pipeline expectations in Lakeflow Spark Declarative Pipelines bring data quality validation directly into your ETL logic. You can **warn** about violations while still processing data, **drop** invalid records to ensure clean outputs, or **fail** pipelines when critical issues occur. The expectation metrics visible in the pipeline UI provide ongoing visibility into data quality trends.

Apply these techniques incrementally in your data engineering workflows. Start with schema enforcement and CHECK constraints for fundamental type and value validation. Add pipeline expectations for streaming workloads where real-time quality monitoring matters. Use quarantine

patterns to isolate problematic records without blocking your main data flows. Together, these practices create data pipelines that your organization can trust.